码农 the code maker

增刊

Linus && Linux



portrait by zendiixx

一生只为寻找欢笑

这个极客很有种

Git为何不用C++开发

Linux发展编年史

一个人加上一百万人的智慧

分布式开发样板

你可能不知道的Shell

iTuring.cn

版权信息

书名：码农增刊 Linus 与 Linux

作者：图灵社区出品

本书由北京图灵文化发展有限公司发行数字版。版权所有，侵权必究。

091507240605ToBeReplacedWithUserId

Linus，一生只为寻找欢笑



作者/ 池建强

池建强，70后程序员，Blogger，《MacTalk·人生元编程》作者。先后任职洪恩软件和用友集团，从事互联网和企业应用软件研发，目前担任瑞友科技IT应用研究院副院长。热爱技术和编码工作，Apple 和 Google 产品重度用户，分享技术，坚持梦想。微信公众号：sagacity-mac

每个人桌面上一台电脑，这曾经是无数计算机先驱的梦想，这个梦想很早就实现了，在 1997 年，乔老师和比老师就说过："比尔，我们共同控制了 100% 的桌面系统市场。"当然乔老师没说的是，比老师控制了 97%，乔老师还不到 3%。时至今日，乔老师走了，比老师颓了，移动终端把传统的 PC 市场冲击得七零八落。普通用户都知道了 Windows、Android、OS X 、iOS、BlackBerry等，但是，他们依然不了解的是另一款在计算机发展史上起到了革命性作用的操作系统：Linux！

当大家使用 Google 搜索时，使用 Kindle 阅读时，使用淘宝购物时，使用 QQ 聊天时，很多人并不知道，支撑这些软件和服务的，是后台成千上万台 Linux 服务器，它们时时刻刻都在进行着忙碌的运算和数据处理，确保数据信息在人、软件和硬件之间安全地流淌。可以这么说，世界上大部分软件和服务都运行在 Linux 操作系统之上，什么云计算、大数据、移动互联网，说起来风起云涌，其实没有 Linux 全得趴窝（微软除外）。

但是，Linux 和它的缔造者林纳斯·托瓦兹一样低调，这么牛逼的一个物件，居然只有程序员知道它的传奇，这不科学！所以我准备在这个系列中写写林纳斯：他是 Linux 和 Git 的缔造者，他是一个传统的黑客，与沃兹一样，少年成名，崇尚自由，一生只为寻找欢笑，他，是一名真正的程序员。

林纳斯在 2001 年出过一本自传，叫做 *Just for Fun*，是他和大卫·戴蒙合著的，当年我有幸读到这本书，了解了很多林纳斯的生平轶事，那时我就琢磨，这个天才已经达到人生的巅峰了吧，结果这位兄台并未停止前进的步伐，转眼就在 2005 年搞出了分布式版本控制系统 Git，目前几乎全世界的程序员都在用 Git 管理他们的代码，著名网站 Github 就是基于 Git 构建的。无论是 Linux 还是 Git，得一即可得天下，结果这哥们以一己之力发起了俩项目，而且都是主力开发人员。最终的结果是，成全了程序员，陶冶了用户，造福了一方百姓。正如林纳斯自己所言：“My name is Linus, and I am your God.”

Linus（一）**——**生命的意义

1969 年末，林纳斯出生于芬兰的赫尔辛基市，算是赶上了 60 后的尾巴。小时候他是个其貌不扬的孩子，除了一个鼻子长得“富丽堂皇”之外乏善可陈。他为了让鼻子看上去小一些，经常戴上眼镜就不愿意摘下来，这个策略和现在的很多大脸女生购买三星的 Galaxy Note 手机有异曲同工之妙。幼时的林纳斯不修边幅邋里邋遢，不怎么费劲数学和物理就学得极好，社交却一塌糊涂，他母亲经常和别人说，这孩子非常好养，只要把他放到一个有电脑的小黑屋里，然后再往里扔点薯条和意大利面，就行了。林纳斯对此表示认同。

林纳斯把年幼的自己定位成 Nerd（书呆子），但是从他的自传里我却感受到了这位天才的有趣之处。他在书的序言里写道：

> 我对生活的意义有一些看法。咱们可以在第1章跟读者说一下生活的意义来钓他们上钩，等他们上钩，并且花钱买书以后，我们再随便扯点别的把剩下的章节糊弄过去。（注：做人要厚道啊。）

关于生命的意义，林纳斯的解释是，有三件事具有生命的意义。它们是你生活当中所有事情的动机。第一是生存，第二是社会秩序，第三是娱乐。生活中所有的事情都是按这个顺序发展的，娱乐之后便一无所有。因此，从某种意义上来说，生活的意义就是要达到第三个阶段。你一旦达到了第三个阶段，就算成功了。但首先要越过前两个阶段。

Linus（二）——天才也疯狂

林纳斯是这么解释的，技术的诞生同样是为了人类的生存，而且是为了让人生活得更好。汽车让人跑得更快，飞机让人飞得更高，互联网让人懂得更多，手机让人通信更快，一旦这些技术成了规模，就要并入社会秩序，然后下一个阶段就是娱乐，别看手机现在就是个打电话的工具，但是很快会进入娱乐阶段……（12年后的今天，手机已经彻头彻尾变成了一个娱乐工具，打电话反而成了附属功能。）

林纳斯说："一切事物都将从生存走向娱乐，但这并不意味着在某个局部地区没有倒退的现象，而且毫无疑问许多地方都有这种情况。有时事物的发展往往是分裂的。"

从这些内容我们可以看出，林纳斯有自己的一套理论，而且能自圆其说，其实每个人都有自己的理论，一件事做或者不做，都是自己说服自己，每一次进步，要么是推翻自己的理论，要么是完善自己的理论。林纳斯在很小的时候就建立了自己的理论领地，那就是数学、物理、逻辑，最后是计算机，所以他绝不是自己描述的 Nerd，而是一个大智若愚的牛娃，就像射雕里的郭靖一样，看着傻，其实比谁都精，脑子里装的都是十年二十年后的事儿。而且林纳斯比郭靖牛的地方是，就一个启蒙老师，还是自己的外公，和郭靖一比，高下立判！林纳斯基本上就是个自学成才的典范。

林纳斯的外公是赫尔辛基大学的一位统计学教授，数学家。他有一台Commodore VIC-20 计算机（Commodore 是与苹果公司同时期的个人电脑公司，曾经创造过一系列辉煌，1994 年破产），这台电脑的主要功能就是没有功能，你唯一能做的事情就是用 Basic 语言在上面编写自己的程序，老爷子当年就是这么做的，比如做一些数学运算和公式计算等。但是老爷子年老眼花，也不愿意打字，于是就把自己的外孙林纳斯放在腿上，让他帮助录入写在纸片上的程序。这种很有场面感的场景一再出现后，林纳斯除了对数学有了初步的认识，同时也把计算机玩得娴熟，很快他就在外公的指导下开始编写自己的程序。

林纳斯用外公的计算机学会了 Basic 语言，并开始编写各种简单有趣的游戏，然后他又发现了 Basic 并不是计算机唯一能理解的语言，在它的下面，还有一种语言是由 0 和 1 组成，可以直接被计算机识别，于是林纳斯又开始用机器码编程，这次他可以控制更多计算机的细节，他与机器变得更加亲密。然后林纳斯就开始上中学了，中学的几年于他而言，其实没有太大变化，因为那些年他几乎都是坐在电脑前面度过的，

在这个阶段，他熟练地掌握了汇编语言。

终于有一天，林纳斯向编程世界挺进的步伐变得缓慢下来，因为他上大学了，原因之一是他必须集中精力读书，原因之二是找不到什么项目去做。还有一件事，林纳斯开始服兵役了，那段时光对他来说是如此特殊：

> 所以当我扛着武器上完了11个月的“体育课”之后，我觉得我余下几十年完全有资格平静快乐、一动不动地度过，以后我唯一的体力活就是敲键盘写代码，或者是抓着一瓶比尔森啤酒享受，这样才叫公平。

Linus（三）**——**改变一生的书籍

终于，让林纳斯痛苦不堪的兵役结束了，除了敲锣打鼓欢庆重生之外，他开始继续拓展自己的编程之路，这时候，生命中最重要的一本书出现了，书的名字叫做《操作系统：设计和实现》，作者是安德鲁·坦尼鲍姆。用林纳斯的原话表述就是“这本书把我推上了生命的高峰”。

那个时代 Unix 已经开发出来了。最早 Unix 是用汇编写的，开发过程中 Unix 的两位创始人肯·汤普逊和丹尼斯·里奇觉得用汇编写程序实在是太苦逼了，男人应该对自己好一点！于是老哥俩决定用高级语言来完成下一个版本，他们首先尝试了Fortran，失败！然后又基于 BCPL（Basic Combined Programming Lanugage）创建了 B 语言，B 语言可以被认为是那个时代的解释型语言，不能直接生成机器码，效率上完全没法满足系统的需求，再次失败！我们都知道，一再失败的情况下总会有一位英雄人物挺身而出，这次是丹尼斯·里奇，他从失败的大坑中爬起来拍拍土抹抹泪，继续对 B 语言进行改造。这次丹尼斯为 B 增加了数据类型，并让 B 语言能够直接编译为机器码，然后又为这门语言起了个极其响亮的名字——New B，读一读神清气爽，念一念气冲云霄，从此一代语言巨星冉冉升起，40年后依然排在兵器排名榜第一位，怎一个牛字了得！当然，丹尼斯可能考虑了十几年后中国人民的感受，把 New B改为了 C 语言，并用 C 语言重新编写了 Unix 的内核，Unix 与 C 从此珠联璧合，长相厮守，再也无法分离。

操作系统、Unix 和 C 语言可以说是林纳斯心目中神山上的三座圣杯，为了至高无上的荣耀，他首先要攀上峰顶，把这三座圣杯捧在手中，然后再琢磨建造自己的宫殿的事儿。在那一年的夏天，林纳斯开始了高强度的阅读和学习，用他的话说就是做了两件事：“一件事是什么都没做，另一件事是读完了 719 页的《操作系统：设计和实现》。那本红色的简装本教科书差不多等于睡在了我的床上。”

林纳斯认为，Unix 是一个简洁、干净的操作系统，在 Unix 上的大部分任务都是通过一些基本操作完成的，这些操作被称为 System Call，顾名思义，这些操作就是你对系统的呼叫，系统通过响应你的呼叫完成工作。比如 fork、clone（创建子进程），比如 open、close、read、write（文件访问）。这些基本的系统调用通过组合可以完成大部分功能。同时，Unix 还提供了极为强大的 IPC（进程间通信）方式：pipe（管道）。很多工作在 GUI（图形界面）软件环境下的读者，最常用的 IPC 操作可能是复制、粘贴、鼠标拖拽，这些操作虽然简单，但是

必须由人来完成，想要自动化就很困难。而这些在 Unix 上实现起来就像大自然一样自然，你只需要在程序之间开辟出一段缓冲区作为管道，然后父进程和子进程就可以通过这个管道实现进程间通信了。举个例子，以前给大家介绍的查找历史命令的脚本，就利用了管道的功能，如下：

```
history | grep apache
```

这行命令的含义就是查找包含 apache 的历史命令，其中特殊字符“ | ”用来告诉命令行解释器 （shell） 将前一个命令的输出通过“管道” 作为接下来的一行命令的输入，就这样，一个简单的进程间通信就完成了。

总之，林纳斯在读完这本书之后，就像郭靖修习了九阴真经全本一样，对机器和代码的世界有了更为透彻的认知，接下来的事情就是等待一个打造传奇的机会。

等待的过程中，林纳斯也没闲着，他又开始编程了。好的程序员对编程的喜爱是溢于言表的，以下摘录一些林纳斯的编程感想：

> 在编程的人看来，编程是世上最有意思的事情了。它要比国际象棋之类的游戏复杂得多，你想要什么规则都可以自己设定。按照你定下的规则，它的结果该是什么，就会是什么。你在电脑上创造出属于自己的新世界，而唯一的限制就是电脑本身的性能，还有，还有一点在今天尤为重要，那就是你自己的能力。比方说造一间树屋吧。你可以造一个带活板门的树屋，这样实用而稳固。不过这样一间仅为坚固实用而造的朴素树屋，和一间为了外观漂亮而巧用树木特点来精心雕琢的树屋比起来，人人都看得出来两者间的差别。造树屋可是一种将艺术和工程融合起来的活儿。编程也是如此，也正是因为这个原因，编程才得以成为一件既有魅力又有价值的活儿。编程时，程序员往往优先考虑的是趣味性、美观性及震撼力，而非实用性。

在代码的世界里，林纳斯就是一个诗人！

Linus（四）——Linux 诞生

Unix始于20世纪 60 年代，在 70 年代得到了迅猛的发展，这时候的林纳斯还躺在祖父公寓里的摇篮里睡大觉，如果不是后来 Unix 王国自乱阵脚，出现阵营分裂和法律纠纷，可能 Linux 系统根本都不会出现。真实的情况是，Unix 浪费了大把的时间和机会，似乎就是为了等待这个大鼻子、头发纷乱的芬兰小子长大，然后一决高下。林纳斯赢得了自己的时间，他一刻不停地磨练自己的技艺，在清晨的微光中练习算法，在赫尔辛基的雪山上编译代码，随时随地地补充粮草和武器。21年之后，林纳斯抚着雪亮的刀锋上路了，他要去追寻属于程序员的最高荣耀。

1991年1月，林纳斯花费了 3500 美元，分期付款购买了一台杂牌组装电脑，内存 4 兆，CPU 33 兆赫，还有一台 14 英寸的显示器，然后又买了 MINIX 操作系统，用 16 张软盘把这个操作系统装到了计算机里。之后，林纳斯又用了一个月的时间，了解了 MINIX 的好和不好，并把这个系统改装成了自己得心应手的“战斗机”，开始了战斗的人生。就是在这台电脑上，催生了 Linux 的初始版本。

Linux 的诞生离不开 MINIX，MINIX 是 Mini Unix 的缩写，是安德鲁·坦尼鲍姆教授编写的迷你版的 Unix 操作系统，源代码可以提供给大学和学生，用于操作系统教学，采用了微内核设计。其中的代码还作为《操作系统：设计与实现》的示例程序，这本书我们在 “Linus（三）”中提到过，给了林纳斯极大的启发。

林纳斯使用了 MINIX 之后，发现这个系统有很多缺陷，比如性能问题、内核问题、文件系统问题，最大的问题是终端仿真器，也就是我之前总提到的 Terminal，登录学校里的 Unix Server 和上网时，林纳斯都需要终端，但是 MINIX 无法满足这个需求。如果普通人遇到这种问题，估计就是发会呆然后洗洗睡了，或者说“你行你上啊”，但林纳斯不是普通人！

他决定抛开 MINIX，从硬件层面开始，重新设计一个终端仿真器。牛人就是不同凡响，这个决定表明了林纳斯需要从 BIOS、CPU 等硬件层面重新开发出一套系统，除此之外，还需要了解如何把信息写入显示器，如何读取键盘输入，如何读写调制解调器，早期储备的汇编语言和 C 语言能力终于派上了用场……

两个月之后，终端仿真器完成，对此林纳斯非常骄傲：

对我这次了不起的壮举，妹妹萨拉倒是知道，不过我把终端仿真程序演示给她看时，她在显示器上看到一长串字母 A 和一长串字母 B，看了大概五秒钟，然后说了一句“挺好的”就转身走了，显然没觉得有什么大不了的。我意识到这确实不像什么了不起的事。只不过，有些东西表面上可能看起来没什么，但背后却包含了大量繁杂的工作。这就好像你把一段刚铺好的柏油公路指给别人看，指望这样就能让他明白你耗费了多少人力物力，这根本不可能。当时是三月份，或者四月吧。不知道彼得盖坦大街上积雪是不是都化成雪泥了——我不知道，也不在乎。大部分时候，我都穿着睡袍，和我那台不招人喜爱的电脑缠绵在一起。窗户上那块厚实的黑窗帘遮得严严实实，把我和阳光隔离开来（外面的世界就更不必说了）。Linux 操作系统就这样开始了，一发不可收拾。林纳斯的当时编程状态是这样的：编程——睡觉——编程——睡觉——编程——吃饭——编程——睡觉——编程——洗澡——编程……

实现了终端仿真器之后，林纳斯马不停蹄，开始添加磁盘驱动和文件系统，那一年林纳斯还在上课，但是课程很简单，他唯一的课外活动就是参加每周三晚的同学聚会，这个长着大鼻子的技术天才，常常会因为担心自己缺乏社交能力和容貌丑陋而失眠，对那时的他来说，唯一有趣的事情就是把驱动程序写出来。于是他咬咬牙对自己说，还得干下去。（看来没有女神的好处就是可以写个操作系统出来，然后把自己叫做上帝。）

随着工作的进展，终端仿真器正在开始向一个操作系统的方向发展，林纳斯显然也看清楚了这一点。

在创造 Linux 的整个过程中，我们没有看到林纳斯使用了什么样的高级工具，估计那时也没有，整个系统基本上是一行行代码敲出来的，纯手工打造，这些先贤的编程功底和效率让我们叹为观止，所以，现在，我决定打开终端，输入 vi，然后键入 to be continued，感受一下林纳斯当年编程的风采……

随着林纳斯不断地敲击键盘，他的终端仿真程序也不停地扩张，从刚开始的小树苗长成了一株盘根错节的大树，树根牢牢地抓住土地，枝丫努力地伸向天空，花朵和果实开始在高远的天空中烁烁发光，所有的细节都在林纳斯的掌控之中。懂行的技术人员都看得出来，这个大鼻子的芬兰小子是准备开发一个操作系统啊。

是年 6 月份，林纳斯基本确定了要开发一个操作系统内核的计划，并开始着手搜集 Unix 操作系统标准的相关资料。1991 年 7 月 3 日，格

林尼治时间上午 10 点钟，林纳斯在 MINIX 新闻组发出了一封求助邮件，寻求有关 POSIX[1] 标准的帮助，他在邮件中写道：

> [1]注：POSIX 全称是“可移植操作系统接口”（Portable Operating System Interface）。IEEE 最初制定 POSIX 标准，是为了提高 UNIX 环境下应用程序的可移植性。随着技术的发展，POSIX 开始不局限于 UNIX 系统，后续的 Linux 和 Windows NT 都部分地遵循了该标准。POSIX 在林纳斯开发的过程中起到了灯塔的作用，直接后果就是 Linux 系统从一开始就走在了正规军的康庄大道上，基本没有跑偏过。Linux 几乎可以适配各种类型的硬件体系结构。

由于我目前正在Minix系统下做一个项目，所以对POSIX标准定义很感兴趣。不知道有没有人能给我提供一个最新版的 POSIX规则（最好是以一种机器可读的格式）？要是能有FTP地址就更好了。
这份公开的邮件是标识 Linux 问世的最早证据。邮件发出后不久，有人就寄来了厚厚的 POSIX 标准，同时赫尔辛基工学院的阿里·莱姆克也对林纳斯的邮件做出了响应，为林纳斯提供了一个 FTP 地址，用来上传他即将完成的操作系统。

标准和 FTP 地址都有了眉目，林纳斯开始实现各种系统调用，以便让 shell 运行起来。这段时间的工作让林纳斯时常感到灰心丧气，看着增加的代码量，工作似乎前进了一大步，但是检验一下功能又仿佛没有任何进展。有时候他还不得不放弃之前的想法和已经完成的代码实现，另辟蹊径从头再来，即使是在天才面前，代码也能让人欢喜让人忧。

终于 shell 已经可以在新的操作系统上工作了，林纳斯开始编写复制（cp）和列表（ls）等程序。shell 程序一旦完成，就好像完成了从 0 到 1 的飞跃，一切都变得无比顺利，林纳斯面前仿佛出现了一条阳关大道，一切都豁然开朗了，他说，要有光，于是就有了光。对于这种状态，林纳斯表示“我很满意”，并且开始用“Linux”称呼这个操作系统。

这种满意非常重要，因为那个夏天林纳斯除了伏在电脑面前噼噼啪啪地敲击键盘，什么都没做。芬兰四月到八月的日子是一年中最美好的时光，人们到布满小岛的海上航行，去海滩上晒日光浴，到夏日小木屋中消遣时光。但是林纳斯，他只是在永无休止地编写程序，忘记了白天和黑夜，黑色的窗帘遮蔽了灿烂的阳光，也遮蔽了外面的世界。他唯一的想法就是，得赶紧把这该死的系统做出来！

1991年8月25日，林纳斯在 MINIX 新闻组上发邮件做了一个调查，想知道大家希望这个新的操作系统具备什么特征。

1991年9月17日，林纳斯把已经完成的新操作系统上传到阿里·莱姆克提供的 FTP 服务器上，并准备用 Freax 作为操作系统的最终代号，结果遭到了阿里·莱姆克的激烈反对。阿里·莱姆克对林纳斯说：

“林哥，您咋会想到用这么变态的名字命名操作系统呢？原来的 Linux 不挺好的嘛！”

“那样不会显得自恋吗？”

“您这样就不对了，操作系统是开天辟地的大事，人民群众都等着用您的名字命名呢，看看他们的眼神，您能辜负他们的期望吗？ Linux 天生不就是用来跟 Unix 遥相呼应的么？这是命，得认！”

“这……那我就不推辞了啊。”

以上为意译，不过基本上和古代皇帝的黄袍加身是一个意思。新的操作系统最终以 Linux 命名，并在 10 年后名扬天下，20 年后统治服务器领域，可谓 Linux 恒久远，Linus 永流传。

Linus（五）——继续前行

Linux 从一诞生就被打上了开源的烙印，这一点对 Linux 的后续发展起到了至关重要的作用。从 1991 年内核 0.01 版本发布，到 1994 年 1.0 版本闪亮登场，世界各地无数的开发者为 Linux 提交了代码，林纳斯为 Linux 建立了讨论组 comp.os.linux，全世界爱好开源和 Linux 的程序员与黑客都在上面讨论问题，他们就像群蜂筑巢一样，不断地通过个体和群体的力量交替推进 Linux 的飞速发展。

林纳斯对自己说：嗯，没有任何东西可以阻挡 Linux 的普及！

这种感觉估计很多程序员都体会过，当你设计的算法得出了正确结果的时候，当你自以为解决了一个海森堡 Bug（Heisenbug，表示不可重现）的时候，当你完成了一段精妙代码的时候，你摘下厚重的眼镜，推开铺满灰尘的书桌，打开办公室唯一的窗户，迎着夕阳把一只废弃的圆珠笔扔出窗外，然后冲着天空大喊：还有谁？这是一种拔剑四顾心茫然的情怀。

林纳斯还不止于此。他不仅单枪匹马写出了 Linux 的内核，而且做出了开源的决定。他把 Linux 放到了互联网上，并且允许那些希望使用和改进它的人们根据开源协议修改和提交源代码。这两点对互联网的影响是极其深远的，估计林纳斯当年也没有想到，当时的两个小小的涟漪，经过时间和空间的放大，十几年后形成了一股互联网巨浪，到现在 Linux 依然处于风口浪尖。

对于 Linux 取得的成功，林纳斯将其归结为是由自己的缺点导致的：

1.我很懒散；
2.我喜欢授权给其他人。

其实这两个所谓的缺点，正是优秀程序员和领导者必备的要素，它们让 Linux 成为世界上最大的开源协作项目，为喜爱 Linux 的人们带来了最美好的技术和应用，现代的互联网几乎是运行在 Linux 之上的，可以说，林纳斯改变了世界，你每一次伐开心后在淘宝上买包包，都有林纳斯贡献的力量！

Linus（六）——来到硅谷

1996 年的春天，Linux 顺利发布了 2.0 版本。是年林纳斯 27 岁，这个芬兰小子已经慢慢厌倦了芬兰平淡无奇的日子和不眠不休的编程生活。对于一个技术天才来说，创造一套新的技术体系就像艺术家完成一个雕像一样，当一块粗砺的岩石在他的亲手打磨下逐渐显山露水，展现出其完美容颜的时候，后续的修修补补会让这些天才产生倦怠的感觉。他们需要更快的剑、更高的山和更强大的对手。尤其是期间林纳斯访问过两次美国之后，这种感觉变得愈发不可阻挡了。

说起来美国确实是个神奇的国度，这样一个移民国家中，居住了各种从不同国度不远万里跨海而来的种族，每个种族无论在基因上还是文化上都具有原来国家的特质，这些特质相互融合与对抗，让这块大陆上的人民更锐意进取，更开放，更自由，他们愿意去追求和接纳美好的事物，最终一不留神把美国搞成了世界文化的大熔炉，而开放的文化和环境又极大地激发了人们的想象力和创造力，近代和现代的科技成果几乎全部源于美国，要么是美国人搞的，要么是外国人在美国搞的。所以有时候我们也不用顾影自怜，嘲笑自己没有国产的操作系统和编程语言，因为其他国家也没有，或很少有，芬兰好不容易出了个天才少年，也没好好珍惜，最终落了个“流落”异国他乡的下场。

林纳斯一到美国就被这块新大陆吸引了，一切都是那么地新鲜和美好，他的感受与你第一次出国后在微信朋友圈发的“天是那么地蓝，云是那么地白”是一样一样的。林纳斯在自传中写道：

> 当时，摩门教的 150 周年纪念活动才刚过了没几年，所以他们主要的教堂都拾掇得干干净净的，白得发亮。在欧洲，由于历经岁月蹉跎，所有的教堂都显得老态龙钟，斑驳陆离。看惯了欧洲的教堂，再一看摩门教堂，这白得发亮的墙让我联想到迪斯尼乐园！它看起来不像是教堂，更像是童话里的城堡。我记得我从金门大桥上徒步走过，边走边望着对岸的马林岬，兴奋极了，巴不得一步走到对面，在群山的怀抱中畅行。不过，等我总算走到马林岬那一边的时候，已经累得走不动了，什么群山和畅行早就抛诸脑后了。当时我肯定料想不到，差不多六年之后，我会坐在海风拂面的马林岬群山顶峰，一边俯瞰着太平洋、旧金山市、旧金山湾、金门大桥以及笼罩着这一 切的雾霭，一边对大卫的录音机诉说这一切。

从美国回到芬兰之后，林纳斯对自己说：“我要去美国。”

当林纳斯透露出自己的就业计划之后，马上有多家公司递来橄榄枝，其中包括著名的 Linux 公司 Red Hat。这种感觉是如此美妙，就像你刚刚掏出一支香烟，面前已是千百个打火机舞动。但是林纳斯本着不加入任何一家 Linux 公司的原则，拒绝了 Red Hat，参加了另一个名不见经传的公司的面试。这家公司叫做 Transmeta，中译名“全美达”，你们可以从维基百科上查到这家公司，不过我打赌，知道这家公司的读者不会超过千分之一。这并不是咱们孤陋寡闻，因为美国人民刚开始也不知道这家公司在干嘛，全美达官网在 1997 年中上线，两年半后网站的建设情况是“This web page is not yet here”，又过了很久人们才从内部员工透露出的一点信息得知，这家公司似乎是搞处理器的。这是我所知道的唯一一家保密措施强过苹果的公司，如果不是林纳斯，这家公司就像是根本没有存在过。

就是这样一家公司，面试了在开源社区名满天下的技术天才、Linux 操作系统的缔造者林纳斯，并且将其招至麾下，一待就是六年。从某种程度上，这六年严重地影响了 Linux 操作系统前行的脚步，因为林纳斯没有足够的时间开发 Linux 了。

虽然根据全美达与林纳斯的协议，他可以继续从事 Linux 的开发，而且他确实也想这么做，比如白天为全美达工作，编写 X86 解释程序，晚上继续 Linux 的伟大事业。不过真实的情况是，晚上丫睡着了……

关于加班和睡眠，林纳斯是这么解释的：

> 有些人喜欢加班加点干活，非得轮上两班、三班甚至四班的活儿不可。我不是那种人。不管是全美达的项目还是 Linux 的任务，我都不会牺牲睡眠时间。好吧，如果你非要听真话，其实是我非常爱睡觉。总之，林纳斯第一次从互联网上消失了，很多悲观的开发者纷纷奔走相告，林纳斯这小子是不是被招安了？丫开始为商业公司干活了，Linux 作为自由软件是不是已经濒临死亡了？每当这时候林纳斯就会出来给大家打打气说，哥还在呢，只不过刚睡醒……

关于林纳斯的这段经历，曾经在硅谷工作过的一位朋友给我提供了一些文字，大意是这样的：

> 每次想起林纳斯这段经历，我都要感慨万千。第一次得知林纳斯虎落硅谷的事是在 2002 年夏天，当地的《水星报》记者先是把

林纳斯大吹一通，然后说他从芬兰老家搬到美国，就职于全美达已五年有余，但 H1 移民仍然停留在劳工卡初级阶段，六年期满就要打道回府了。

当时这份报纸的读者大概有一半人有 H1 经历，然后这一半人里的一半都知道 Linux 是啥东东，但是从未听说过全美达是何方神圣，这货居然把一代技术英雄扣在那儿为一个名不见经传的小资本家作苦力，导制全球开源事业停滞不前，真是胆大包天啊！于是很多读者跑到《水星报》去说，像林纳斯这样的天才愿意移民到美国，布什亲自开飞机去接都不为过，怎么可以被移民局压了五年呢……

还好，林纳斯在 2003 年离开了这个叫做“全美达”的公司，受聘于开放源代码开发实验室（OSDL: Open Source Development Labs, Inc），重新统领开源世界的各路英豪，全力开发 Linux 内核，Linux 再次焕发出勃勃生机，这一次，它要引领的是互联网的技术浪潮……

Linus（七）——关于财富

林纳斯对待财富的态度就是“视金钱为粪土”，是真的粪土。

那种默然的态度让人感觉非常可怕。当一个人随便动动手挂挂名签个字就能获取上千万美元的时候，他依然和自己的妻女一家人挤在圣克拉拉一栋两层楼的公寓套房里，过着一个普通程序员的生活，同时不断改进已经遍布全球的 Linux，这是什么精神？这是毫不利己专门利人的国际主义战士的精神。写到这我不禁想起了绿茵场上的冰王子博格坎普，当他接到几十米外的长传，用标志性的慢速停球过掉扑上来的后卫，轻扣，过掉另一个后卫，颠球，闪过最后的防守，面对守门员的时候不是大力抽射和仰天长啸，而是把球搓出一道完美的抛物线，球越过门将，缓缓落入网窝，然后博格坎普，低着头慢慢地走开，留给对手的是优雅与实用并世无双的技艺，和令人绝望的背影！

默然的感觉，懂了撒？

很多程序员创业成功或跟随创业成功之后，自以为功成身退，最早扔掉的就是代码和编译器，然后购豪宅当天使满世界贴旅游照片，你们感受一下，这个境界是完全不可同日而语的。（请勿对号入座，如有误伤，必是友军所为。）

事实上林纳斯在拿到第一笔真正的财富之前，一直处于日子紧巴巴的状态。当时另两位带头大哥比尔·盖茨和史蒂夫·乔布斯早已名满天下家私万贯，同时有大量的技术人员、商人和公司通过 Linux 及其相关技术获取了巨额财富，对此，林纳斯的态度是：“和我有毛关系。”他似乎对一大群才气不高的编程人员能够享受到大笔的财富并不在意。这种情况一直持续到所有的有识之士都坐不住了：林纳斯，你再也不能这样下去了！

伦敦的一位企业家希望林纳斯在他羽翼未丰的 Linux 公司做个董事会成员，报酬是一千万美元。林纳斯说，不用。企业家惊呆了，当他喃喃自语“卧槽你特么知道一千万美元是啥概念吗”的时候，林纳斯已默默走远。

Red Hat 公司为了感谢林纳斯的卓越贡献，为他提供了一些期权，林纳斯的回复同样是：“不用了，我不会给你独家的授权许可的。”Red Hat 的人差点疯掉：“林爷期权您就收着吧，我们什么都不要行了吧？”“唔这样啊，那就放这吧。”这就是林纳斯！

正是这笔期权让林纳斯收获了第一笔巨额财富，因为 Red Hat 1999 年 8 月 11 日在纳斯达克上市了。林纳斯先是意识到自己从身无分文突

然变成了拥有五十万美元的土豪，然后是一百万， 五百万，林纳斯终于变得亢奋起来，原来期权也是钱啊！终于不用再为生计发愁了，对着这个事情，林纳斯的定义是： 我真是最幸运的家伙！

事实上林纳斯从来没有想过 Linux 能够获得如此巨大的成功。他只是为了自己方便写了一个操作系统内核并想借此获得一点回报而已。“假如我事先知道了要做到如 Linux 这般成功需要做多少基础和琐碎工作的话，那我肯定会相当沮丧的。这意味着你首先要非常优秀，并且你所做的大部分决定都导致了正确的结果。”

任何理智的人在登山之前凝望着高耸入云的山峰和崎岖艰险的山路时，都会陷于沮丧之中。解决办法就是先迈出第一步再说，然后，但行好事，莫问前程。

Linux 不仅给林纳斯带来了名声和财富，同时给大众带去了巨大的好处。年轻一代中最聪明的程序员和黑客都在使用 Linux 的产品，正是开放的 Linux 给这些天才的程序员带去了巨大的创作热情和喜悦，他们在 Linux 平台上完成了一个又一个杰出的作品，这些技术形成的生产力，对互联网的发展起到了巨大的推动作用，直到今天。

Linus（八）——巨星碰撞

在 Linux 出现之前，桌面操作系统的市场基本上是由比老师和乔老师控制的，虽然乔老师控制得少了一些。Linux 出现之后，桌面操作系统的格局并没有太大变化，但是服务器端市场的变化却是翻天覆地的。原本比尔希望通过 Windows NT 和 Server 系列在服务器领域复制桌面操作系统的辉煌，从而千秋万载，一统江湖。然而，世界的发展永远是多元的，没人能通过一己之力改变历史发展的多维性，比尔·盖茨也不行。于是 Linux 出现了，并以星星之火可以燎原之势一举拿下服务器操作系统的半壁江山。

一方是商业公司和封闭的策略，另一方是自由软件和开放的协议，这场战争一开始支持率就是一边倒的，林纳斯就像对抗风车的堂·吉诃德，但是他自己不仅没有遍体鳞伤，还在没怎么亲自出场的情况下把微软这个软件风车搞得狼狈不堪，这种情况发生在现实生活中绝对是老百姓喜闻乐见的，林纳斯成了自由软件世界里的英雄和领袖，但也就此与微软结下了世仇，比尔和林纳斯许下了永世不相见的誓言。

有些加盟微软的朋友告诉林纳斯，他们曾见到他的头像被钉在了微软公司的飞镖靶心上。林纳斯对此的评价是：一定是我的大鼻子太好瞄准了。林纳斯与另一位业界巨头苹果之间就没这么激进了，毕竟 Linux 和 OS X 师出同门，都是从老前辈 Unix 那儿毕业的，坐在一起还能唠唠家常，事实上林纳斯和乔布斯确实有过一次历史性的会面。

林纳斯来到硅谷不久，就收到了一封来自乔老师秘书的邮件，邮件中写道："听闻阁下光临硅谷，蓬荜生辉，老乔不才，重回苹果，以期振昔日之雄风，如得阁下相助，必将如猛虎加之羽翼而翱翔四海，天下可得。期待会面。"（当然是意译。）

林纳斯看完之后不明白乔布斯要干什么，只是觉得很厉害的样子。毕竟林纳斯还坐在外公腿上拨弄电脑键盘的时候，苹果的沃兹已经纯手动打造出苹果的第一代个人电脑 Apple I 了。林纳斯决定去见一下儿时的偶像，并了解一下苹果的新操作系统。两代科技巨星的会面被安排在苹果公司位于"无限循环"大道的总部，乔布斯带着原 Next 公司技术总监艾维·特凡尼安（Mach 之父）接见了林纳斯，双方进行了友好而亲切的会谈，然后会谈的结果和某国常规会谈一样，就是没有结果。

其时乔布斯十年放逐回归苹果，举手投足已是大宗师气势。他对林纳斯说，我大苹果虽然现在看起来有点颓，不过海盗精神永存，我们已经准备好重新起航了。目前个人电脑领域仍然只有两个玩家：微软和苹

果。如果 Linux 和苹果能够珠联璧合，那一切将是最好的安排，所有的开源爱好者都能够用上优雅与极客并存的 MacLinux 了。然后 Mach 之父艾维·特凡尼安向林纳斯详细介绍了整合 Mach 和 Linux 内核作为 OS X 混合内核的计划，之后庞大的 OS X 体系将构建在 Mach 和 Linux 内核的基础之上。同时乔老师表示，基于 Mach 和 Linux 的内核系统将采用开源的方式运作，这样全世界的开源爱好者都可以为 Mac 和 Linux 开发程序。

这几乎是一个完美的双赢方案，乔老师都被自己描绘的蓝图打动了，永远年轻，永远他妈的热泪盈眶！谁能拒绝苹果公司和乔布斯如此完美的邀请呢？

林纳斯能！

乔布斯认为自己的扭曲现实力场加上苹果巨大的市场潜力一定会让林纳斯怦然心动，没想到这个芬兰小子在计算机面前待久了，水米油盐不进，任凭乔布斯口吐莲花，我自巍然不动。首先林纳斯对 Mach 就不感冒，他认为 Mach 几乎犯下了所有的设计错误，它让系统变得复杂而效率低下；其次林纳斯觉得乔布斯可能没意识到，Linux 的潜在用户要比苹果系统多；第三林纳斯乐观地认为，虽然 Linux 的目标不是占领桌面操作系统，但是显然“我们很快就能做到这一点了”。所以林纳斯当时的反应是：

> 我不在乎啊！为什么我就应该对苹果的事情感兴趣？我就是不感兴趣，我就是觉得苹果没意思。我的人生目标可不是为了抢占台式机市场啊。（当然，虽然 Linux 马上就要做到这点了，但这从来就不是我的目标。）

现在看来，林纳斯当时对 Linux 在桌面操作系统的前景过于乐观了，虽然他天纵奇才桀骜不驯，但是也无法预测到 OS X 和 iOS 在十年后引领移动开发的浪潮。不过即使知道 OS X 未来的大发展，心高气傲的林纳斯也不会接受苹果的收编，因为 Linux 一直是独立和自由的软件图腾。

无论如何，这次非正式的会谈没有达成任何实质性的效果，但是对后来的 IT 格局产生了巨大的影响。苹果不再关注 Linux，而是转向了 BSD。2001 年苹果任命 FreeBSD 的发起人之一、老牌 BSD 黑客乔丹·哈伯德为 BSD 技术经理，后升为 Unix 技术总监，负责 OS X 操作系统底层核心 Darwin 的研发，最终，Mach 与 BSD 技术整合在一起，形成了混合内核。另外，苹果开始觉得开源项目也不是那么靠谱，后续他们先

后研发并开源了优秀的编译器项目 LLVM 和 Clang，一举替换了整条 GCC 编译链，为 OS X 和 iOS 的性能优化和语言特性提供了巨大的帮助。这也算是苹果对那些牛叉哄哄的开源人士的回击：看，我们也可以做开源，而且比你们做得好。

Linux 则继续在开源、独立、自由的方式下一路狂奔，虽然在桌面操作系统领域的成就乏善可陈，但是在服务器端大放异彩，目前几乎整个互联网都是运行在 Linux 及其衍生产品之上的，可以说没有 Linux，互联网不可能得到如此迅猛的发展。十年以后，移动互联网时代来临。OS X 上长出了 iOS，Linux 上则诞生了 Android，这两个移动开发领域的双子星都有一个老祖宗，那就是 Unix。一次话不投机的会谈让 OS X 和 Linux 分道扬镳，在十几年后的今天，它们又以一种不同的方式相见了，世界永远都是多元的，可能冥冥中自有天意吧。

Linus（九）——Linus 和 Git

很多人在完成了类似 Linux 这样宏伟的软件产品之后，基本上就止步不前了。但是林纳斯却从未停歇创新的脚步。2003 年加入开放源代码开发实验室之后，林纳斯重新全职投入 Linux 内核的研发，并开始酝酿自己的另一个跨时代的产品。

2002 年，Linux 内核开发团队开始采用 BitKepper 作为代码版本管理工具。BitKeeper 是一套分布式的版本管理工具，它满足了 Linux 内核开发的技术需求。但是 BitKeeper 只是暂时对 Linux 等开源软件团队免费，并不是自由软件。2005 年 BitMover 公司不再免费赞助 Linux 开发团队。对此林纳斯表示非常遗憾，但遗憾之后他并没有自怨自艾伤心落泪，而是愤怒地与其他几个小伙伴花了几个星期完成了一套新的分布式代码管理工具，命名为 Git。两个月之后，Git 发布了官方版本，并在不同的项目中应用，自由软件社区给予了 Git 广泛的支持。

与 SVN 和 CVS 等软件不同的是，Git 更关注文件的整体性是否有改变，Git 更像一个文件系统，它允许开发者在本地获取各种数据，而不是随时都需要连接服务器。Git 的最大的特点就是离线分布式代码管理，速度飞快，适合管理大型项目，难以置信的非线性分支管理。

2005 年 Git 发布之后，技术日臻成熟，很多大公司都开始采用 Git 管理自己的项目代码，2008年2月 Github 公司基于 Git 构建了协作式源代码托管网站 Github，目前该网站是这个星球上最大的源代码集散地，几乎所有的优秀代码都托管在 Github 上。

Git 已经成为程序员使用最多的源代码管理工具！

对于 Git 的成功，林纳斯表示：

> Git 的设计其实很简单，它有一个稳定而合理的数据结构。事实上，我强烈建议围绕着数据来设计代码，而不是反其道而行之，我觉得这可能就是 Git 如此成功的原因。坏程序员总是担心他们的代码，而优秀的程序员则会担心数据结构和它们之间的关系。

从 Git 诞生到今天已经有 9 个年头了，Git 始终没有背离其设计的初衷：高性能、简单的设计、非线性高并发分支的支持和完全的分布式。

对于林纳斯来说，Git 现在是他的主要消遣工具之一。他很喜欢在 Git 上编程的感觉，因为再也不用担心锁定问题、安全问题和网络问题，这种感觉真是太美妙了！

我们继续期待林纳斯的第三个伟大的作品！

Linus（十）——生活的意义

林纳斯认为生活的全部意义就在于：生存，社会交往和寻找乐趣。因为我们所做的一切事情，最终似乎都是为了我们自己的乐趣。而进化作为主线始终贯穿其中。

林纳斯对进化的理解是：

> 你知道在整个太阳系，人类已知的最复杂的工程是什么吗？不是 Linux，不是Solaris，也不是你的汽车。是你，还有我。想想你和我都是怎么来的——不是什么超复杂的设计，没错，凭运气。除了运气，还有：
>
> - 通过分享“源代码”实现自由的可用性和授粉机制，生物学家把它称作DNA。
> - 毫不手软的用户环境把我们不好的版本轻易地替换成更好的可执行版本，从而使种群更加优秀（生物学家把这叫做“适者生存”）。
> - 大量的无方向的并行开发（试错法）。
>
> 我从未如此严肃过：我们人类永远都无法复制出比我们自身更复杂的个体，而自然选择却不假思索地做到了。不要低估适者生存的力量。不要错误地认为你可以做出比大量的平行试错反馈环更好的设计，那样就太抬举你的智力水平了。说实话，太阳照常升起，这和任何人的工程技巧或者编程风格都没有关系。

林纳斯一生只为寻找欢笑，但是他却取得了无数的成就和荣誉：

- 1997年，在芬兰赫尔辛基大学计算机科学系，林纳斯接受了他的硕士学位。两年后，他在斯德哥尔摩大学接受名誉博士学位，并在 2000 年在他的母校获得了同样的荣誉。
- 1998 年，林纳斯接受了电子前哨基金会先锋奖。
- 2004 年，林纳斯被《时代周刊》杂志选为世界上最有影响力的人之一。
- 2006 年，《时代周刊》杂志欧洲版评选林纳斯为过去 60 年最有革命性的英雄人物之一。
- 2012 年 4 月 20 日，林纳斯被宣布成为两位获奖者之一，

和山中伸弥共同获得当年的千禧技术奖。该奖被公认为相当于在技术领域的诺贝尔奖。

- 2012 年 4 月 23 日，林纳斯进入互联网协会（Internet Society，ISOC）的网络名人堂。

林纳斯憎恶分明，经常口不择言，比如他对 C++ 的评价是：C++ 是一门糟糕的语言。而且有一群不合格的程序员在使用 C++，他们让它变得更糟糕了。他对自己的两个产品命名的解释是：我是个自大的混蛋，我所有的项目都以我的名字来命名。开始是Linux， 然后是 Git（英国俚语，饭桶的意思）。

不过我最喜欢林纳斯说过的一句话是：Talk is cheap, show me the code。他一直用自己的编程人生诠释着这句话。2006 年的时候，Linux 内核代码的 2% 依然是林纳斯完成的，他是代码贡献最多的人之一（是年 37 岁） 。到了 2012 年，他对内核的贡献主要是合并代码，编程变少了，但是他依然对是否将新代码并入到 Linux 内核具有最终决定权。

林纳斯用自己精彩的编程人生和对自由软件的热爱演绎了现代社会中一个书呆子的胜利。如果你爱一个人，就让他去编程吧；如果你恨一个人，就让他去编程吧。代码让我们欢笑，也让我们忧伤，让我们沉默，也让我们高歌。对于程序员来说，代码是这个世界上最美妙的音乐。会编程的孩子，都是好孩子！

本文参考资料：

1. 只是为了好玩：Linux之父林纳斯自传
2. 维基百科相关资料
3. Mac OS X 背后的故事（二）——Linus Torvalds的短视

这个极客很有种



作者/ Linus Tovalds

林纳斯•托瓦兹，当今世界最著名的程序员、黑客，开源操作系统Linux之父。生于芬兰，毕业于赫尔辛基大学，1997年～2003年任职于美国加州硅谷的全美达公司，现受聘于开放源代码开发实验室，全力开发Linux内核。2004年，他被《时代周刊》评为世界最有影响力的人之一。

你是否曾在一个盛夏的夜晚仰面朝天，遥望星空，弄不明白自己为什么会来到这个世界上？身在何处？这辈子又能做些什么？

没错，这种事嘛，我也没干过。

但最起码，我对生活的意义有自己的理论，关于这个世界，关于一切的一切，或者起码是关于那个被我称为“生活”的子集。

我对生活的看法，不是从某夜仰望星空，在月朗星稀的夜晚里感叹宇宙的浩瀚无边得来的，而是从我为某一次演讲的准备中得来的。一旦你因为某样事物而出了名，人们就会对你寄予厚望，坚信你对困扰了人类几百万年但又彼此没什么关联的各种知识能够产生一些深刻的洞察力。而且，他们还想要你对着一群全然陌生的观众分享你的真知灼见。

不，这种寄望并没有多大意义。我之所以搞Linux系统，仅仅因为我是个技术极客，而不是因为我擅长在公共场合露脸，更不是因为我擅长从哲学角度不着边际地思考问题。但好在人生中也没几件事是有很大意义的，这样一来对于别人的寄望我也没什么好抱怨的。

说回我们刚才的话题。

启发我去思考生活意义的，是一场伯克利大学举行的名叫“Webrush”的活动，他们邀请我去参加。要在平常我都懒得搭理，但他们的邀请是通过芬兰驻美国领事馆发给我的，作为一名爱国人士（或是出于因为受不了冰天雪地的小事就移居外国的那点愧疚感），于是我便傻乎乎地用芬兰语说：“好的，我一定出席。”

那次座谈会，应该没有谁会指望我阐释生活的意义吧，起码我自己没有。但会议是和网络化社会的生活有关的，而我是作为互联网和芬兰的双重代表出席的。多亏了诺基亚（任何一个芬兰人都会告诉你，那是全世界最大、最好及最美丽的公司），芬兰对电子通信有着难以割舍的依赖，正处在“网络化社会”的时代。我们之前提过，芬兰这个国家拥有手机的数量比本国人口的数量还要多，而且目前科学家们致力于研究合适的方法，以便能在人出生的时候就把手机缝在身上。

就这样，我坐在那儿，思考着关于“电子通讯”这个话题该说些什么。噢对了，有一点我忘了说，除了我之外，现场的其他嘉宾都是一些准备上台讨论技术的哲学家。这毕竟是伯克利大学啊，他们学校最重视两样东西了：伯克利的政治学专业和伯克利的哲学家。

所以，管他呢。反正，他们学校要是让在场的哲学家讨论技术的问题，那我何不作为技术专家来讨论哲学问题呢？谁敢说我没种？也许他们会说我是愚蠢到家了（嘿，他们可能真的说了），但谁敢说我是胆小鬼？

不会的，这个极客很有种。

所以我就顺着我的话题一个劲儿地思考，好在第二天站在讲台上有话说。（我总是这样不见棺材不掉泪，不到最后一刻是不会准备演讲稿的。所以演讲前的那个晚上，你就会见到我为隔天的演讲担心不已。）我就这么苦苦挣扎着，盯着“电子通讯”左思右想，电子通讯到底是怎么回事呢？诺基亚及其他所有的通讯公司最终又会如何发展呢？

想来想去，我觉得我能干得最好的还是解释生活的意义。

其实不能算是“意义”吧。更准确地说应该是生活的法则，没准以后还能称为“林纳斯法则”。它与物理学上的热力学第二定律差不多，但不是用来解释宇宙的退化次序，而是生命的进化。

我说的这个“进化论”可不是达尔文的那个进化论，这是完全不同的东西。针对Webrush的听众，我更感兴趣的是社会是如何进化的，以及人类是如何从工业社会过渡到信息社会的，接下来会发生什么？为什么会这样？我想让自己的演讲变得动听，要是能在一次座谈会的时间内说服在场的观众就更好了。每个人在那场活动中都有各自的议程安排，而那天我的安排是与两位著名哲学家进行小组讨论，并且活着回来。

那么，社会为什么会进步呢？其驱动因素是什么，真的是科技吗？科技似乎已经是公认的因素了。蒸汽机的发明使欧洲进入了工业社会，最终诺基亚和手机又使我们进入信息社会，真的是这样吗？大概只要是哲学家就会这么认为吧，而且他们似乎对科技如何改变社会的这个过程很感兴趣。

我呢，作为一个技术专家，却认为科技其实什么也没推动。应该是社会改变了科技，而不是反过来。科技只不过限制了我们做某件事能达到的程度，以及我们完成某件事的最低成本。

科技，和它创造的所有产品一样，就目前而言都是愚蠢到家的东西。科技唯一有意思的地方，就是你能用它来干些什么，而且它背后的驱动因素实际上就是人类真正的需要和兴趣。如今人与人之间面对面的沟通已经很少了，因为我们有了新的沟通方式。新的沟通方式能够得到广泛的普及，正是因为人类生来就爱嚼舌头，他们想要交流；就算没有沟通的渠道，他们也会去创造。诺基亚就是这么来的。

因此我认为，要了解社会的进化，我们必须知道是什么在真正地驱动人类。是金钱，是成功，还是性？到底是什么动机从根本上驱使着人类做他们正在做的事？

最明显的动机，大家应该都没有异议吧，就是：生存。毕竟生存定义了生活，人是要生存的。这可不像盲目地跟从热力学第二定律那么简单了，而是要在一个对构成了生命基础的复杂事物与规则充满敌意的世界里生存。总之，生存就是头号动机。

为了给其他动机排序，我们必须考虑一下它们和那个纯粹的生存动机相差有多远。问题不在于“你会不会为了钱去杀人？”，而在于“你会不会为了钱去死？”。答案显然是否定的。因此，我们可以很有把握地把金钱从“根本动机”那张单子上剔除掉。

但显然的，世界上还真有人类愿意为之献出生命的东西。有许多关于人类，或者是动物的英雄故事，他们愿意为了更伟大的事情而牺牲自己的性命。因此单靠生存动机并不足以推动整个人类社会的进步。

我在伯克利大学的演讲中提到了其他几个动机，这些动机很简单，在讨论小组里面也没有什么异议。至少有人对我说的动机表示赞同（或者，他们表示赞同只是出于礼貌，因为不好拂了领事馆的面子）。人类愿意为之牺牲性命的事物不多，但我敢打包票社会关系肯定是其中之一。

人们因为社会关系这个动机而献出生命的例子实在太多了，从文学作品中的罗密欧与朱丽叶（他们献出生命，不是因为他们想得到诸如性这种愚蠢的东西，而是他们宁愿死去，也不想失去彼此间特殊的社会关系），到愿意为了自己的国家和家庭牺牲生命的爱国战士（对他们而言，这些就是社会关系）。所以，把“社会关系”列为第二点动机吧。

第三个动机也是最后一个，就是“娱乐”。这个听起来或许是陈词滥调，但它作为一个强有力的动机绝对是毋庸置疑的。每天都有一些人死在了他们仅仅是为了好玩才做的一些事情上。就比如说，坐在飞机里本

来平安无事的，有些人偏要跳出机舱玩滑翔，就是因为他们觉得有趣。

娱乐不一定非得是指某些老掉牙的东西。它可以是一场国际象棋比赛，也可以是一次对“世界如何运转”这个问题的思考，还可以是对新世界的好奇和探索。娱乐能驱使人们为了从外太空看清地球的面貌，而自愿坐在一支装着数亿磅烈性炸药的狭窄火箭里，所以娱乐当然可以称作“动机”。

总结一下，我说的三点动机依次是：生存，人在社会秩序中的位置，以及娱乐。就是这些动机驱使我们做我们正在做的事。除了这三点，其他的那些所谓动机，顶多是像社会学家所说的“突现行为”（emergent behavior），这种行为基本都是出自一些更加简单的规则。

但仅仅说“驱使人类的只有这些动机”是不够的。单靠这些，还不足以形成有关生命的理论。有意思的是，这三个动机有内在的先后次序，只要是有生命存在的地方，它们就会依次显现和发挥作用。受这三个动机驱使的，可不光只有我们人类，在其他生命体的生存和进化过程中，它们也会依次显现和发挥作用。

生存，社交，获得乐趣，这就是它们显现和发生作用的次序。这也就是我们选择*Just for Fun*作为书名的原因。因为似乎我们所做的一切到头来都是为了寻求乐趣，至少如果我们目前可以算是进化得足够高级的话，那我就有资格说，事情确实是这样的。

你不相信？

看看我们是如何把动物分为“低等”动物和“高等”动物的吧，这都是从生存的角度出发的。进化的规模越大、等级越高，就越有机会建立社会模式，蚂蚁就是如此。虽然它们在进化排行榜上排名靠后，却也有着森严的社会模式，并且社会秩序阶段终究会向娱乐阶段发展。不过，蚂蚁确实不会做玩弄食物的事……但猫可经常这样做。当然了，蚂蚁也不享受性。

没错，性就是一个浅显的（也是愉快的）例子。我并没有说它本质上是几个根本动机之一，但是作为人类的一项基本行为，性经历了整个生命的进化历程，所以它是一个非常好的例子。毫无疑问，这玩意刚开始只是一种纯粹的生存手段。毕竟，从生存的角度来说，连植物都有“性生活”。而且在数十亿年前的某个阶段，对于单细胞动物来说，性还真是一种再单纯不过的生存手段，不过这些单细胞动物通过缓慢的进化最终变成极客和其他人类。同样毫无疑问的是，性在很久以前就已经从一种纯粹的生存手段进化成一种特殊的社会行为了。举行结婚仪式或者其他许多为了获得性福而准备的仪式，可不是只是人类才有的行为。

想想沙丘鹤的“求偶舞蹈”，顺便说一下，它们一辈子就一个伴侣。是为了生存下去才要寻找伴侣的。实际上，各种生物社会每天消耗在求偶仪式上的能量简直不计其数，它们的动机再简单不过了，就是为了繁殖下一代。

性的娱乐方面呢？也是这样，我向你保证。不是只有人类才懂得用性创造欢乐，这个星球上进化得最好的物种似乎都懂得从性中获得最大限度的愉悦，这些现象大概不是巧合吧。

从生存到社会行为，从社会行为到娱乐，这一进化无处不在。就拿战争来说吧。战争最初明显是一种生存手段，为了占据水源，唯一的办法就是把挡道的家伙干掉，因为那家伙正好也想把水源占为己有。长久以来战争就是一种维护社会秩序的手段。随着CNN电视台的出现，战争成了娱乐。你接受也好不接受也罢，这些进化都是不可避免的。

文明本身也遵循着相同的模式。最初这是一种集中群体的合作和力量来确保生存的手段。这不是人类特有的。绝大多数动物，甚至连植物都会建立社群，以便更好地互相帮助并生存下去。有意思的是，社会就是这样，都是从以生存为本逐步向更高级的社会化演变；所有的人类文明，最终都在建造更宽广更耐用的公路和更好的通讯频道的行为，这些都是为了更好地实现社会化。

最后，文明也不可避免地走向了娱乐阶段。看看罗马帝国，他们不仅以修建马路和建立强大的社会秩序而闻名，而且，尤其是后来，以娱乐生活的发达程度而闻名。

或者看看今天的美国。电影和电脑游戏行业难道不是正在将美国引入娱乐化社会吗？这点应该不会有人质疑吧。这些东西以前还是利基市场里的东西，现在它们已成为这个世界上最富有的国家中最大型的产业了。

而且，作为一个技术专家，让我觉得有意思的是，这种进化模式正在我们创造的技术世界里重演。我们把现代技术的早期阶段称为“工业时代”，但实际上这个阶段应该叫做“工业的生存时代”。技术，直到不久前，还只是为了生存得更好些，例如为了更快更好地织出布匹，为了更快捷地运送商品。这就是所有技术得以开发的最初动机。

我们将当前的这个时代称之为“信息时代”。这个时代较之以往发生了很大的转变。在这个时代里，科技被用于通讯和传播信息，这是相当社会化的行为，已不仅仅是为了更好地生存了。不只是互联网，事实上，许多技术都在朝着这个方向努力，这使得互联网成为我们这个时代的巨大路标：在工业化国家里，一旦有了互联网，就意味着人们已经将生存视为理所当然的事情。并且一夜之间，科技的第二阶段到来了，这

是一个巨大而激动人心的进步：通讯技术社会化的阶段已经到来，于是运用技术不仅是为了更好地生存，而且是社会生活中不可或缺的一部分。

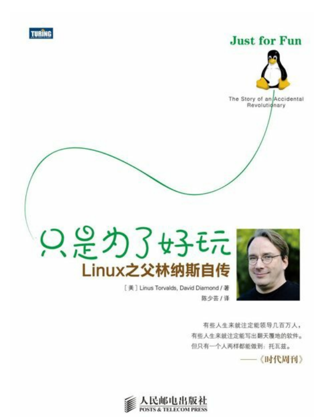
当然了，终极目标对我们来说仍是海市蜃楼。进入信息化社会之后，就应该向娱乐化社会过渡了。在娱乐化社会里，全天候无间断的互联网和无线通讯已被视为理所当然，也就不再引人注目了。到那时，思科公司已成往事，世界将由迪士尼公司说了算。或许这个时代离我们已经不再遥远了。

说了这么多，这些到底都意味着什么呢？其实也没有什么。毕竟，我对“生活的意义”的这点看法并不能真正指引你去做应该做的事，它顶多是说：“是的，你应该奋斗，但生活的最终目的是为了享受乐趣。”

从某些方面来说，生活的意义的确如此，这就解释了为什么人们愿意甚至渴望在互联网上为Linux这样的项目工作。Linux能同时满足两个动机，于我，于他人，都是这样。生存是理所当然的，而Linux能使人们通过挑战智力获得乐趣，又能通过参与开发工作而获得一种被社会需要的满足感。我们也许并没有多少机会见面，但电邮也不只是干巴巴地交流信息，在交流的过程中邮件已经成为友谊和其他社会关系的纽带。

也许这也意味着，如果我们有幸与宇宙中另一智能生命体相遇，他们说的第一句话可能不是“带我去见你们的领导”，而是会说：“哥们，你们的派对开得正欢嘛！”

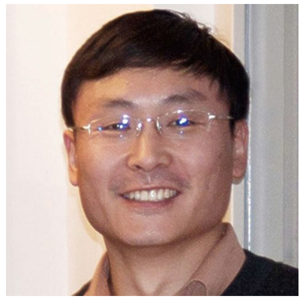
当然了，我说得不一定对。



2001年Linus与人合著了他的自传《只是为了好玩》，也是唯一一本关于他的传记。这部自传风趣幽默，充满极客风格，向读者展示了林纳斯如何用自己精彩的编程人生和对自由软件的热爱演绎现代社会中一个书呆子的胜利。13年后我们重新翻译出版这位最有影响力发明家一的传奇经历，期待与更多人一起探寻如何以“一切为了好玩”的态度对待技

术、软件和人生…… 本文节选自《只是为了好玩》。

Git为何不用C++开发——Linus痛贬C++的经典邮件



译者/ 李松峰

图灵QA部主任。2006年起投身翻译，出版过译著20余部，包括《JavaScript高级程序设计》、《简约至上》等畅销书。2008年进入出版业，从事技术图书编辑和审稿工作。

From: Linus Torvalds (torvalds (at) linux-foundation.org)

Subject: Re: [RFC] Convert builin-mailinfo.c to use The Better String Library.

Newsgroups: gmane.comp.version-control.git

Date: 2007-09-06 17:50:28 GMT (4 years, 8 weeks, 1 day, 20 hours and 54 minutes ago)

2007-9-5，周三，Dmitry Kakurin 写道：

>从第一次看到Git的源代码起，就有两件事如鲠在喉

>1.纯C而没有C++。不知道为啥。

>2.别提可移植，那都是胡说。

你才是胡说八道。

C++是一门很恶心的语言。但更恶心的还是有一群下三滥的程序员在用它，结果弄出来的都是一堆堆的排泄物。坦率地说，就凭可以把那些C++蠢货挡在门外这一条，就足以证明应该使用C。

换句话说：使用C证明我们神经没有毛病。我知道Miles Bader曾开玩笑说要“气死你”，但实际上是当真的。我已经想清楚了，不管是谁，只要他认为这个项目使用C++会比使用C更好，我都更想“呸”他一口，好让他滚得远远儿地，少来干扰我的项目。

C++会导致最最差劲的设计。总是要使用STL、Boost等等这些臭狗屎，还美其名曰“精巧”的语言库，倒是“帮”你编程了，可是却会导致：

- 这些狗屎歇菜时无穷无尽的烦恼（说什么STL，特别是Boost既稳定又好移植，全都是胡扯，胡扯好玩吗？）
- 低效的抽象编程模型，而且两年后你才发现某些抽象并不是很有效率，可你的全部代码都是在那些对象模型上构建起来的，非重写代码不能解决问题。

这么说吧，要想弄出一个不错的、高效的、系统级的而且可移植的C++方案，唯一的结果就是把自己限制住，而这些在C里面都可以信手拈来。把项目限制用C来开发，意味人们不会搞砸它，也意味着很多程序员实际上都理解底层的问题，不会用白痴的“对象模型”把问题搞得无解。

所以真的很抱歉，但对于Git来说，效率一直都是主要目标，C++的“优势”只能酿成巨大的错误。而让我们有机会对那些看不到这一点的人吐唾沫只不过是它的另一个很大的附加优势。

如果你想要一个C++写的VCS，试试Monotone。真的，它使用“真正的数据库”，用了很多“出色的面向对象库”，还用了“出色的C++抽象”。恕我直言，这些设计决策虽然可以让那些CS迷们神魂颠倒，但结果必然是极其恶心、难以维护的大杂烩。

但是，我相信你一定会选它而不是Git。

Linus

原文链接：http://article.gmane.org/gmane.comp.version-control.git/57918

Linus Torvalds访谈：我已经不读代码了

作者/ Glyn Moody

Glyn Moody是一位技术书籍作家。他曾著有《反叛的代码：Linux和开源的革命》（*Rebel Code: Linux and the Open Source Revolution*），这是一本纪录开源软件运动的著名黑客访谈集。



我在Linux发展的早期就有幸采访过Linus，那是1996年的时候，当时他还住在赫尔辛基（在旧版"连线"杂志的人物中可以读到这篇文章）。当时正是他人生的关键时期，既是他个人的（他第一个孩子刚刚出生），也是事业上的。他当时正要加入Transmeta这家芯片设计公司，这事虽然没成，但是他却移居到美国，直至今日。

如今他来欧洲的机会已经很少了，而我借着他将在巴塞罗那的LinuxCon欧洲2012发表演讲的机会再次采访了他，我们回顾了Linux内核发展的一些关键点以及上次交谈以后社区的进展。

Glyn Moody: 回顾过去15年，你觉得在内核发展方面有什么关键性的进展？

Linus Torvalds: 对于我来说，头等大事就是我们所有关于扩展性的工作。我们已经过了3、4台CPU就过得去的时期，现在也许4000台才是个坎儿，但是大多数时候内核并不是瓶颈。如果你的工作负载相对稳定，我们的扩展性已经做得相当好。这是在付出很多努力之后的成果。

有几百台CPU的时候，SGI在扩展性方面就能派上用场。他们最开始的补丁无法合并，我们无法在普通PC上利用他们的工作成果，因为它

的基础构造就是为了服务于上千台CPU的。当你只有一两台的时候，就太昂贵了。

我最担忧的其实是为大型机准备的高性能内核，这些的源代码和普通内核是分开的。人们为了保证代码基的简洁性已经花费了很多精力，这样在编译的时候就可以说：我需要能为4000台CPU服务的内核，由此生成代码，如果你拒绝的话，那我需要可以为2台CPU服务的内核，同时同样的源码来编译。

回顾起来这是相当重要的一点，因为这让源代码变得更好了。所有SGI的努力以及统一源代码的成果都被清洗了一遍，因为它在一百台CPU上行不通，所以这是无法避免的。而且，这也让内核的可维护性变得更好了。现在对于台式机来说，8到16台CPU都是很正常的，以前扩展到8台就是个门槛，现在则完全不在话下。

但是也有其他重要的事情。我们在另外一端也花费了数年的时间，手机使用者已经清楚地意识到他们的hack丑陋不堪，特别是那些在ARM微处理器上为了节省电量所做的努力。我们花费数年时间在总体的电源管理上，以及移动电话适用人群关注的一些设备，我们努力将这些融入到内核中来。我们用了5年时间才使得电源管理系统派上用场，因为这涉及到一套完整的体系。

通常来讲如果添加一个设备，并不会影响到其余的内核，但是电源管理是个例外，它会影响到所有的上千台设备的驱动。它影响到核心功能，如关闭CPU，调度机，以及包括虚拟机在内的所有东西。

还不只是影响，它甚至还有毁掉所有东西的可能，这些都让人很痛苦。花费了那么多时间，我们其实只向前挪动了两步，还倒退了一步，因为虽然这步是一个显著的提高但是却毁坏了我们的机器。所以退回的一步完全用来修补被破坏了的设备。

从现实角度考虑，大部分的发布都只是关于驱程序的工作。这其实挺无聊的，驱动程序的本质就没什么意思，它只是为了支持其他芯片集，毕竟，这些东西都是内核的柴米油盐。大部分的内核其实就是驱动程序，所有我们看起来又酷又炫的工作，在为了支持新硬件而做的努力面前都黯然失色了。

架构、社区，以及**Linus**的角色

Glyn Moody: 为了支持新硬件有什么架构方面的重大变革吗？

Linus Torvalds: USB的堆栈被重新改写了几次，就是因为一些新用例的产生，我们觉得原来的USB子系统没有把这些考虑进来。USB3.0也需要新的主控制器来支持，而这些东西带来的差异足以让我们更改核心堆栈，以使得其在不同版本上都能使用。这些改变不只限于USB，也包

括PCI，PCI变成了PCIe，这时候热插拔又出现了。

这就是传统Linux和传统Unix的一点不同。如果你有一个Unix工作站，启动它之后，它就不会变，因为你不会增加设备。现在人们对于增加USB设备都习以为常了，但是过去可不是这样的。为了实现热插拔设备，我们几乎更改了所有的基础构造，为了与时俱进。

Glyn Moody: 内核社区现在的发展怎么样了？

Linus Torvalds: 过去，我们的社区是比较扁平的。我也不知道什么时候开始了这样的改变，过去只有我和大约50个左右的开发者，现在这里划分了好多等级。补丁要到我这层估计要经过4层左右的人。我们大概3个月会发布一次，每次发布都会牵涉到1000人左右。而这些人中的一半大概只提交了一行不太重要的代码，有些人就是这么工作的，而且以后也再没啥贡献，这其实也挺好的。你想想如果涉及1000人的话，而有一些只是打酱油的，不可能要求我把每个人提交的补丁都过一遍。我没这么多时间和这么多人打交道。

有些人比较擅长驱动器，有些人又特别擅长于特定的驱动器领域，他们就要和开发独立驱动器或者提交补丁的人打交道。我所看到的补丁大概就要经过这些层，虽然有时候是4个，但是经常是2个人。

Glyn Moody: 那你的角色现在是什么样的？

Linus Torvalds: 最大的改变莫过于我不再读代码了。当一个补丁已经至少经过了两个人的时候，我就可以看着这个补丁说：你们所有工作都白费了，我在这个层面进行微管理——说实在的，我不想这么做，我也没有能力这么做。

所以大多数时候，它们（补丁）经过子系统的主要维护人员，这些人都是我信赖的人，我和他们共同工作长达5年，10年，甚至15年，所以我连代码都不用看。他们会给我一个很高等级的概述，告诉我这就是做出的变更。他们可能用5行内容告诉我什么改变了，然后给我一个diffstat告诉我在文档中改变了15行左右，在文档中改变25行意味着diffstat可能要有上百行，因为有几百个文档都发生了改变。所以我不用亲自面对代码，就可以说这些变更可以执行，顺便说一句，我很信任你。然后这些就会执行起来。

Glyn Moody: 那你现在的角色是？

Linus Torvalds: 从广义上说，我是在管理人。并不是后勤方面——我并不给任何人发工资，也不担心他们会进入到不应该的硬件中。大部分时候我在人们产生争议或摩擦的时候介入，或者当bug出现的时候。

bug无时无刻不在产生，而人们不知道该找谁，所以他们就会把bug报告发送给Linux内核邮件列表，几乎没有人能面面俱到地阅读这些。

如果在邮件列表中没有人能搞明白的话，他们就会开始轰炸我，并且说：这机器怎么不工作了呢？虽然我已经不读代码了，但是我知道该找谁，所以最终我会作为这个bug报告的牵线人出现。这就是我现在干的，整天整天地读邮件。这也挺好的，我也乐于做这些事，但是这和我曾经所做的已经大相径庭了。

其他工具？平板和云

Glyn Moody: 这是否意味着你会写出其他像Git一样的工具，用于管理人，而非代码？

Linus Torvalds: 我目前没有这样的打算。也许会有新的工具产生，但是实际讲，我现在所做的都是和人打交道。我们有用于人员管理的工具。这些工具的作用就是：我们知道是这部分代码出了问题，谁最后接触了这些代码，谁是这个子系统的维护人员，因为整个系统牵涉的人太多，不用自动化系统来处理这件事是行不通的。但是多数时候工作内容都是与人接触，不同人有不同的工作方法，过度依赖自动化装置对于人来说也是不合适的。

我们现在干得不错。10年前那种痛苦纠结的时刻已经没有了。那都是由于以前的扁平结构造成的，而现在我们改进了工具，改变了工作流。并不是只有我，整个内核没有一个人不在这种工作流程下工作。

我会收到海量的邮件，但是我并没有被这些邮件淹没。我喜欢在旅行的时候通过手机读邮件。甚至在休息的时候我也会读邮件，因为其中90%我只需要把它们存档就可以了。除此之外不用做任何事，抄送给我的邮件很多，告知我现在发生了什么事。所以在旅行时，我就可以做90%的工作，甚至都不用电脑。晚上回到宾馆，我会把剩下的10%解决掉。

Glyn Moody: 16年前，你说过你做的很多事都是由外在世界所推进的，现在大家对平板和移动这么关注，这些对内核的发展有什么影响？

Linus Torvalds: 在平板领域，很大一块问题来源于电源管理，因为平板比手机要大。它们的电池更大，人们期待平板的电池待机更长也想要更大的屏幕，而这两点是矛盾的。在内核方面，从硬件角度和使用角度来说，平板和手机是非常相近的，而这方面我们是有些经验的，因为毕竟有Android。

平板的用户界面有一些让我们纠结的问题——但是那些和内核离得比较远。在手机上，浏览器通常不是一个完整的浏览器——过去都是专用的移动浏览器；但是在平板上，大家都期待有一个完整的浏览器。所以关于平板的大部分问题都是在用户空间方面的。在手机内核方面我们确实有很多问题，但是大部分问题也是平板共有的。

Glyn Moody: 关于云计算呢？这些对内核有什么影响呢？

Linus Torvalds: 最大的影响在于，在云计算条件下，甚至服务器端，人们都更加意识到能量功耗的问题。过去，所有关于能量功耗的问题都是在嵌入式人员以及手机这里发生的，而在最近3-4年，服务器端的工作人员也意识到能量的问题。因为太多的机器被聚在一起；而经常来讲，使用还有峰值。如果研究亚马逊的话，你就会发现他们峰值使用量是通常状态下的好几个数量级。比如说在亚马逊的销售方面，11月末和12月的时候，圣诞节的一个月以前，他们的工作量和整年的工作量差不多。问题是他们要为这样的峰值扩展硬件基建，而平时只能用到这些能力的十分之一。所以不用每时每刻都使用能量是很重要的，因为电力对于这些巨大的服务器来说是很大的消耗。

亚马逊、**Google**，以及邮件

Glyn Moody: 亚马逊加入到内核的工作中来了吗？

Linus Torvalds: 亚马逊不是一个很好的例子，其实Google更合适，因为他们雇了一堆内核工程师。多数时间，Google自己就能解决问题。我觉得亚马逊面临的问题更多的来自于标准配件。事实上他们已经改变了构建硬件的方法——他们现在自己有自己的硬件参考设计。他们以前都是从惠普和戴尔那里购买硬件的，但是到头来如果你发现你需要购置一万台机器的时候，还是自己设计起来更简单，你就可以和设备生产厂商说：我就要台这样的。但是他们只是最近才开始这样做。

我不知道是亚马逊有些落伍了，还是Google太以科技为核心了。亚马逊在用户空间方面做了不少工作，他们的内核是很标准的。而Google的工作核心更偏重于内核，他们做了自己的文件系统。他们以前也是自己做硬盘的驱动，因为他们对此有一些特殊的要求。

Glyn Moody: Google在内核方面的工作对你们有用吗？

Linus Torvalds: 有几年时间，甚至是5到10年时间，Google都是一个黑洞。他们自己雇用内核工程师，而且会突然消失得杳无音信。他们在Google内部工作，然后从此再也没有人听说关于这些人的事，因为他们做的都是专属于Google的东西，而Google又没有什么相关的反馈。

这些现在都已经极大地改变了，可能是因为Google在我们发布的2.4版上呆的时间太长了。他们在上面干了好几年，因为为自己专属硬件定做的内部修改实在是太多了，所以升级内核对于他们来说是个大问题。一部分也因为整个的Android工程，他们也想在上游区域更加活跃一些。

现在他们更活跃了，人也不会突然消失了。事实说明内核也更好了，因为很多问题都是细枝末节上的而不再是巨大的缺口了。他们现在

已经平易近人多了，只是在标准内核上做一些小改动，而不是为了适应自己的基础设备做一些大手术。

Glyn Moody: 最后，你说你用很多时间来回邮件，作为一个曾经的出类拔萃的黑客，你为此担心吗？

Linus Torvalds: 我觉得还好。我现在不再像以前那样写很多代码了。从另一方面看，其实有一些类型的编程我也不希望再做了。当我20岁的时候我喜欢玩设备驱动。如果这辈子我都不用再写设备驱动了，我会觉得很幸福。就像头疼少一点总不是坏事。

但是我很喜欢Git，这个可有意思多了。我开始整个设计，然后在用户空间编程，而我有15年没干这事了，但是感觉很棒，超简单。我什么也不用担心，有无尽的堆栈，分配内存也刚刚好。但是在内核空间，我就要担心锁定问题、安全问题，以及硬件问题。玩Git，是一种放松。但是最近我也有点厌倦了。

另外还有一个我还在做的项目是关于潜水电脑的。在kernel.org站点，曾经发生过一次非法闯入。对于维护人员来说，那可真是苦事一桩啊，最后FBI也给扯进来了，想看看到底发生了什么。结果两个月来，内核一点进展都没有，虽然人们仍然可以开发内核，但是大家聚在一起的主站点宕机了，而很多核心内核人员要花很多时间检查是否有人闯入过自己的机器。当时大家都有点被害幻想症了。

所以有几个月的时间我的工作都不存在了，因为我需要整合不同人的工作，而我们的整合服务器宕机了。所以我又开始玩我的潜水记录软件，无聊嘛，这个还蛮有意思的。所以到头来我还是编程的，最后，我总会回到内核上。

英文原文：Interview: Linus Torvalds – I don't read code any more

只是为了好玩？

作者/ David Diamond

大卫•戴蒙 职业撰稿人，曾为《纽约时报》、《连线》、《美国周末》及许多刊物供稿。与Linus Tovalds合著有《只是为了好玩：Linux之父林纳斯自传》。



背景：《只是为了好玩？》这本书最开始写于一辆新款黑色福特车上，当时这辆车正在加利福尼亚州中部谷地的某个位置，沿着5号州际公路朝南行驶。林纳斯•托瓦兹与妻子朵芙•托瓦兹和他们的女儿，帕特丽夏和丹妮拉，在一个外人的陪同下，驱车560公里前往洛杉矶，准备去参观动物园和宜家家居商店。

大卫：现在我正在思考一个最基本的问题，这个问题挺重要的。你想借这本书表达什么？

林纳斯：这个嘛，我想阐释生活的意义。

朵芙：林纳斯，你没忘记给车子加油吧？

林纳斯：我对生活的意义有一些看法。咱们可以在第1章跟读者说一下生活的意义来钓他们上钩，等他们上钩，并且花钱买书以后，我们再随便扯点别的把剩下的章节糊弄过去。

大卫：对啊，这倒是个好点子！有人说，自从人类起源，人就一直被两个问题困扰着：第一，“生活的意义是什么？”第二，“到头来，口

袋里攒下来的那点儿闲钱该往哪儿花？”

林纳斯：第一个问题我已经有答案了。

大卫：有答案了？是什么？

林纳斯：基本上这个答案是干脆而漂亮的。答案本身不会解释生活的意义，但会直接告诉你生活中会发生什么。有三件事对生活是有意义的，它们是生活中所有事情的动机——包括你做的所有事和任何一个生命体会做的事：第一是生存，第二是社会秩序，第三是娱乐。生活中所有的事都遵循着这个顺序，娱乐之后就再无其他。所以从某种意义上说，生活的意义就是要你达到第三个阶段。一旦达到了第三阶段，这辈子你就算成功了。但是你得先超越前两个阶段。

大卫：这个你得详细再解释一下。

帕特丽夏：爸爸，我们能不能停下来买巧克力冰淇淋？我好想吃巧克力冰淇淋！

朵芙：不行啊小宝贝儿，你得等一等。待会儿咱们停下来便便的时候，再给你买冰淇淋。

林纳斯：我简单举几个例子吧，好让你大概明白我的意思。最好理解的就是性了。它最开始只是延续生命的方式，但后来变成了一种社会行为，这也是人们结婚的原因。再后来它就成了一种娱乐。

帕特丽夏：那现在我要便便了。

大卫：性怎么个娱乐法？

林纳斯：好吧，有点儿对牛弹琴了。我换一个例子吧。

大卫：不，还是说回性吧。

林纳斯：它是从另外一个层面来说的……

大卫（自言自语）：噢，这娱乐说的是参与其中的娱乐，而不是置身事外来观赏的娱乐啊。好吧，我懂了。

林纳斯：……如果从生物学的角度来看待人们对性的幻想，那就是另外一个层面的问题——性最初是怎么来的呢？最初它是作为一种生存手段，和娱乐一点儿都不沾边，性只是两个人融合起来生存的手段。算了，性这个例子我们先放下吧。

大卫：不，不，我觉得这就够讲整整一章的了。

林纳斯：我们来说说战争吧。最初，战争显然是为了生存，你得先和别的大块头抢水源；后来，你又得和他抢老婆。之后战争的目的就是建立社会秩序了。在中世纪还没到之前，战争就已经是这样了。

大卫：就是说，战争是建立社会秩序的手段之一。

林纳斯：对，战争也使个人成为社会秩序的一部分。才没有人本身会去在乎什么社会秩序，他们只关心自己在那个秩序里的地位。这就跟

母鸡在乎的是它在禽鸟里的啄序高低是一个道理。
大卫：那现在战争就成了一种娱乐？
林纳斯：没错。
大卫：可能那些看电视战争节目的人才会觉得战争有意思吧！
林纳斯：电脑游戏，战争游戏，CNN电视台，都是。这么说吧，引起战争的原因常常很有娱乐性，人们也常用娱乐的眼光看待战争。像刚才说的，性行为很多时候也是为了娱乐。当然了，延续生命的那部分动机还是在的，尤其当你是天主教徒的时候，对吧？但即使你是天主教徒，有时候想到性，也会包含娱乐性的需要吧。所以这不一定只是纯粹的娱乐。对所有的事物来说，一部分动机可能是生存，一部分可能是社会秩序，剩下的可能是娱乐。你看，技术就是这样。技术发明最初也是为了生存，后来就不仅仅是为了生存下来，而是为了更好地生存。人们为了从井里打水，才发明了风车……
大卫：取火也是。
林纳斯：对，技术在这些层面依旧是为了生存，还没有达到社会秩序和娱乐阶段。
大卫：那技术是怎么进入到社会秩序阶段的呢？
林纳斯：这么说吧，实际上大部分工业化都是生存需要，或者说是为了生存得更好。就说生产汽车吧，这就意味着要生产更快更好看的汽车。但后来就到了社会秩序的阶段。随后我们就有了电话，到了某种程度，还有了电视。早期的许多电视节目主要是为了给观众洗脑，电台也一样。正是为了社会秩序，很多国家才会从投资电台入手。
大卫：就是为了建立和维护社会秩序……
林纳斯：没错，然后它就超越了社会秩序的阶段。现如今啊，电视节目主要是为了娱乐。而且你瞧瞧现在这些移动电话，虽然大体还处在社会秩序阶段，但它也正朝着娱乐阶段走呢。
大卫：那么技术的未来会是什么样的呢？我们早已超越了生存阶段，现在处于社会阶段，对吗？
林纳斯：对。以前所有技术都是为了使生活更便捷，比如说为了能更快地到达目的地，为了能买到更便宜的东西，为了能住上更好的房子等。那么今天的信息技术又有什么不同呢？人与人之间都建立了联系又会怎么样，还有什么能做的呢？当然了，人与人之间的联系可以更加紧密，但这本质上和以前没有什么不一样。那么，信息技术究竟会把我们引向哪儿呢？我觉得吧，接下来就得踏入娱乐阶段了。
大卫：也就是说，所有的事物最终都会演变成娱乐……
林纳斯：这多少也解释了Linux系统成功的原因。想想这三个动机，

第一是生存，有电脑的人当然没有生存问题。坦白说，如果你拥有一台电脑，那你肯定不必为食物之类的生计发愁了。第二是社会秩序，Linux系统可以说是对建立社会秩序起了积极作用的，尤其是对那些成天缩在他们自己小隔间的极客们来说。

大卫：你在Comdex博览会上说的那一番话很巧妙，当时你说Linux系统的开发是一个全球性的团队运动。所以，基本上是你让社会秩序阶段成为可能了，老兄。

林纳斯：Linux项目确实是个好例子。这说明了人们热爱团队运动，尤其是热爱作为团队一员参与到其中去。

大卫：是啊，每天待在电脑前，你应该很希望不是自己一个人在瞎忙活，而是作为大团队的一份子，什么团队都行。

林纳斯：这就组成了社会秩序，就像其他的团队运动一样。想象一支足球队里的队员，尤其是高中足球队。Linux的社会层面真的是非常非常重要，但Linux系统也有娱乐的层面，而这种娱乐可能有钱也买不到。在生存阶段，钱通常是非常强烈的动机，因为用钱来换取生存是非常容易的，钱能够轻易地换到生存所需。但当你到了娱乐的阶段，金钱就……

大卫：金钱就没有用处了？

林纳斯：不，不是没用，显然你可以花钱去看电影、买车和度假。你能花钱买很多东西来改善生活。

朵芙：林纳斯，我们得给丹妮拉换尿片了，帕特丽夏也得上厕所了。我还想要杯卡布奇诺。你说这儿能找到星巴克吗？我们到哪儿了？

大卫（抬头）：从空中的气味推断，我想我们应该快到金城了。

林纳斯：我们现在说的范围有些太广了，我们说的不仅仅是人，连生活都包括在内了。跟熵定律差不离吧，根据“生活的熵定律”，虽然所有事都会从生存阶段最终进入娱乐阶段，倒也不是说不可能会在哪个环节出现倒退，很多时候倒退是很常见的。有时候这个过程甚至会直接崩溃。

大卫：但是作为一个体系，应该是一直朝着一个方向走吧……

林纳斯：是朝着一个方向走，但不是所有事物都是同步的嘛。因此基本上来说，性爱已经成为娱乐了，战争还在朝着娱乐的方向走，而技术已经很有娱乐的味道了。新事物最初的动机都是为了生存。比如说太空旅行，是吧？它可能开始只是为了生存，后来就开始为社会秩序服务，接着就成了一种娱乐。我们可以把文明看作是一种膜拜形式。是这样的，它们的诞生模式是一模一样的。文明的诞生最开始是为了生存，后来人们为了更好地生存而聚居在一起，这就建立了社会结构，最终文

明就成了一种纯粹的娱乐。好吧，也不能说是纯娱乐，可成为娱乐也没有什么不好啊。古希腊人最出名的就是有严谨的社会秩序，但他们也有不少娱乐活动。谁都知道那个时候他们有全世界最伟大的哲学家。

大卫：好吧，那么这些和生活的意义有什么联系呢？

林纳斯：其实没有什么联系，这些只是说……好吧，这倒是个问题。

大卫：这点小小的联系，就是你接下来要好好想的。

帕特丽夏：妈妈快看，外面有奶牛。

林纳斯：所以说，你要是搞清楚了生活的走向，那么很显然，你的生活目标就是要促进生活朝这个方向走。而且这走向还不是单一的一个走向，你做的任何事都是这些走向中的一部分。你也可以这样问自己："我能做些什么为这个社会添砖加瓦呢？"你知道自己是这个社会的一员，知道社会是在朝着那个方向走，那么你就能推着它往前走。

朵芙（捏鼻子）：这儿怎么那么臭啊！

林纳斯：归根结底，咱们就是为了好玩。那咱们不妨坐着好好放松放松，享受我们的旅途吧。

大卫：就是为了好玩（Just for fun）？



2001年Linus与人合著了他的自传《只是为了好玩》，也是唯一一本关于他的传记。这部自传风趣幽默，充满极客风格，向读者展示了林纳斯如何用自己精彩的编程人生和对自由软件的热爱演绎现代社会中一个书呆子的胜利。13年后我们重新翻译出版这位最有影响力发明家一的传奇经历，期待与更多人一起探寻如何以"一切为了好玩"的态度对待技术、软件和人生…… 本文节选自《只是为了好玩》。

Linux发展编年史

1991

9月17日：21 岁的芬兰学生林纳斯·托瓦兹在网上发布开源操作系统 Linux 0.01。

1992

1月5日：Linux 0.12 版本随同采用 GPL 许可证的版权声明发布，使得开源的 Linux 商用成为可能。林纳斯在后来的一次访谈中称：“让 Linux 遵守GPL 绝对是我干过的最正确的事。”

1993

6月17日：Slackware Linux 由帕特里克·沃尔克丁发布。Slackware 被认为是第一个取得广泛成功的Linux 发行版，它现在还在使用。

8月16日：伊恩·默多克发布了第一个 Debian Linux 的发行版。Debian是最有影响力的 Linux 发行版之一，是 Ubuntu 等众多发行版的鼻祖。

8月19日：马特·韦尔什写的 Linux Installation and Getting Started 第1版出版，这是第一本关于Linux的书籍。

1994

3月14日：Linux 1.0 发布，代码量达 17 万余行，Linux 用户超过 10 万。

11月3日 ： Red Hat Linux套件出售。此后 Red Hat 一路发展成为全球最重要的Linux和开源技术提供商，在纳斯达克上市，银行现金高达29亿美元，成为开源商业模式典范。

1996

5月9日：林纳斯在网上征集企鹅图案作为 Linux 品牌标识，平面设计师拉里·尤因创造的黄背黄脚蹼的名为 Tux 的小企鹅入选。企鹅是林纳斯最喜欢的小动物。

6月9日：Linux 2.0 版发布。这是第一个在单系统中支持多处理器的稳定内核版本，也支持更多的处理器类型。Linux 进入实用阶段，全球有约 350 万人使用。

1997

1月9日： 第一个“Linux 病毒”Bliss被发现。Bliss 不危害系统安全，它依赖于人们用特权干蠢事来感染系统，然后提醒用户只安装从可靠站点下载的可以验证数字签名的软件，并且安装之前一定要先验证签名。

6月：大片《泰坦尼克号》使用 Linux 操作系统制作特效。

1998

5月1日：基于 Linux的 Google 搜索引擎面世。

10月：微软在法国发布了反 Linux 公开信。这表明微软公司开始将Linux视作对手来对待。

12月4日：一份来自 IDC 的报告称 1998 年 Linux 的出货量至少上升了200%，市场占有率上升至少 150%。Linux 的市场占有率为 17%，并且以其他任何操作系统无法企及的速度增长着。

1999

3月：第一届 LinuxWorld 大会召开，象征Linux 时代到来。

7月：Intel 公司启动对 Linux 的支持服务。这可以视作 Linux 真正成为服务器操作系统一员的重要里程碑。

2000

1月： Sun公司在Linux 压力下宣布 Solaris 8 降低售价。

3月11日：摩托罗拉公司宣布发行 HA Linux。这个发行版专注于通信应用领域，对系统不关机连续运行时间要求非常高。它还包括了热交换能力和支持 i386和 PowerPC 架构。

3月23日：爱立信发布基于 Linux 的触屏手机 Screen Phone HS210。

2001

1月3日：美国国家安全局以 GPL 许可证发布了 SELinux。SELinux 提供了标准Unix 权限管理系统以外的另一层安全检查。

2004

10月20日：Ubuntu 以一个不同寻常的版本号 4.10 和怪异的版本代号 Warty Warthog（长满疙瘩的非洲疣猪）问世。用这个版本号是因为发布日期是 2004 年 10 月。Ubuntu 的开发由 Cannonical 公司主导，它虽然不是内核的主要贡献者，然而对于Linux的台式机和笔记本电脑的普及起到了重要作用。

2007

8月8日：Linux 基金会成立，目的是赞助 Linux 创始人 Linus 的工作。基金会得到了包括 IBM、Oracle 等公司以及来自世界各地的开发者的支持。

11月5日：Google 公司发布 Android 移动平台。

2009

1月29日：纽约时报称“现在预计有超过1000万人在运行Ubuntu系统”。

2011

5月11日：Google 发布基于 Linux 内核的云操作系统 Chrome OS。

6月21日：Linus Torvalds 发布 Linux 3.0 版本。

2013

12月13日： Valve 公司发布基于 Linux 的视频游戏控制台操作系统 SteamOS。

第三个系统Linux：一个人加上一百万人的智慧

作者 /Mike Gancarz

Mike Gancarz 是美国佐治亚州亚特兰大市的一名应用开发顾问。他的团队使用 Linux、Unix 和 Java 工具，为金融服务行业开发出多个获奖的成像解决方案。作为 Unix 应用程序设计专家，他不遗余力地推广 Unix 已达二十多年。作为开发出 X Window System 的团队成员，Mike Gancarz 还始创了一些至今仍应用在 Linux 上的最新窗口管理器中的可用性观念。Mike 曾经在 DEC 公司位于新罕布什尔州纳舒厄城的 Unix 工程开发项目组工作，主持了将 Unix 命令和程序移植到 64 位 Alpha 处理器的工作。他的首部著作 The Unix Philosophy（Digital Press, 1995）令数以万计的技术人员受益，他也是《Linux/Unix 设计思想》的作者。



人类只具备创建三个系统的能力。不管如何努力，无论为之奋斗的时间有多久，人类最终都会意识到想要打破这个规律只是徒劳。人类根本无法建立第四个系统。只有自欺欺人的家伙才不相信这个铁律。

为什么只有三个？这是个难以回答的问题。人们或许能从科学、哲学和宗教的观点出发，得到一些猜测性的理论。每个理论都会对这种情况给出一个貌似合理的解释。但是，最简单的解释可能就是，人类的系统设计过程就像人类自身一样，都必须要经历三个生命阶段：未成年、成年和老年。

在未成年阶段，人们普遍充满活力。他们就像是街区里新来的孩子：浑身活力四射，渴望受到关注，而且显示出无穷潜力。在一个人从

未成年阶段发展到成熟阶段的同时，他会成长为对这个世界更加有用的人。他们的职业生涯逐渐成形，与他人的长期合作关系得到发展，在世俗事务中的影响力越来越大。这个人开始给人留下深刻印象——好的坏的或是其他。待到年老的时候，这个人会丧失掉年少时的许多机理能力。而且，随着体格机能的逐渐衰退，其对世俗的影响也会消退，职业生涯都将成为往昔记忆。人们开始抗拒这种改变。然后，沉淀下来的就是那些基于人生经历的宝贵智慧。

人类设计的系统也经历着同样的发展阶段。每个系统都具备与人类生命发展阶段所对应的那些特性。所有的系统都遵循着这条发展路径：从未成年开始，过渡到成熟阶段，直到以老年作为终结。

正如一些人无法活到天年一样，总有一些系统到达不了成熟期。通常，这是由外部环境所决定的。开发计划也许会改变；项目经费可能会被撤销；潜在客户或改变主意，决定购买别家供应商的产品。任何这些因素都有可能导致系统中途夭折。然而，在正常情况下，人们还是可以引领这些系统经历所有的三个发展阶段。为了阐明这个道理，我们将这些阶段称为人类的三个系统。

在背水一战的情况下，人类创建了“第一个系统”

通常情况下开发人员担负着巨大压力，他们必须要赶在截止日期前交付项目，或是满足其他一些时间紧迫的需求。种种压力会激发出他内心灵感创意的火花。最终，在他深思熟虑良久，并在脑海中反复琢磨他的构想之后，这一点火花变成了一小团火焰。工作仍在继续。他的创意本能开始占据上风。灵感的火焰变得越发明亮。

某一刻，他会意识到一部分构想并不只局限在“达成目标”这一范围内。他感觉自己好像在反复思量一些更重要的事物。此时，如果他认为这些构想可以提供更合理的解决方案，那么他就会逐渐淡忘最早制定的目标。

“专家”使用“第一个系统”验证过的想法来创建“第二个系统”

“第一个系统”早期的成功深深地吸引了一些人，他们积极参与进来并希望将自己的名字与这个系统紧密相连，从而获得这样那样的回报。每个人都想和成功产品搭上关系。

这个自封的“专家”群体通常包括许多对“第一个系统”颇有微词的批评家。这些人为自己没能参与“第一个系统”的设计工作而深感懊丧，他们发泄着对“第一个系统”创始人的不满，并声称自己完全可以做得更好。有时候他们是对的。他们确实可能在系统设计某些特定方面做得更好。在重新设计“第一个系统”中几个基本算法的时候，他们的专业知识派上了用场。但请记住：“第一个系统”的设计者（们）并没有足够时间

去把事情做好，而这些专家中的许多人一来知道什么是正确的做法，二来也有充足时间和资源来做好它。

“第三个系统”由那些为“第二个系统”所累的人们创建

经年累月之后，有些人开始发现，这“第二个系统”并不像它吹嘘的那样神乎其神。他们意识到它会吞噬系统的资源，运行起来也非常缓慢。它的确是由专家设计的，只不过其中一些人是自封的“砖家”。虽然“第二个系统”努力去满足每个人的需要，可实质上它并达不到任何人的要求。

不久，“第二个系统”便使许多人精疲力尽。他们认识到所谓的“令你别无他求的系统”只不过是众多系统的一个。它催生了太多的用户组，连处于技术领域外围的人士都开始讲授研讨会。这些人并不像早期激发这股热潮的人们那么才华横溢，因此教学质量一路下滑。夸夸其谈充斥着这些用户会议，外围人士看到之前的专家早已声名鹊起，所以才希望能效仿这些人的派头。只是，他们来得太晚了。

人们都在窃窃私语地交换意见，也许这“第二个系统”并非那么完美，最后，这些小声的传言变成了广大用户发出的集体呐喊。每个人开始怀疑是否会有一个更好的解决办法。直到此时，全世界都做好了迎接“第三个系统”的准备。在与“第二个系统”的对抗中，“第三个系统”诞生了。

另一方面，他们这些“吃不到葡萄”的专家会对别人来之不易的成就大倒酸水。这里面大有一些NIH的意味，也就是曾经流行的“非我发明”综合征。虽然其中许多人其实也有能力去建立“第一个系统”，他们只是没能抢得先机而已。他们参与“第二个系统”的开发工作不是为了找乐子，而是希望借此机会用自己的设计机制来取代“第一个系统”的机制，从而“改善原有设计师明显的业余尝试”，为自己获得专业口碑。

这种态度往往会引起“第一个系统”设计者的愤怒。偶尔，他们也会奋力反击。流行X Window Sytem系统的先驱Bob Scheifler就曾不客气地回应过认为他的早期设计风格过于随意的批评：“如果不喜欢，你完全可以随意编写你自认为具备行业标准的窗口系统。”

Linux既是“第三个系统”，又是“第二个系统”

Linux的出现正逢20世纪80年代末的动荡期，当时Unix处于嘈杂喧嚣的“第二个系统”阶段。“第二个系统综合征”的迹象比比皆是，而Unix社区里，AT＆T和伯克利这些主要派别正为哪一方实施的系统可以主宰Unix世界而争论不休。此外，人们还成立了诸如开放软件基金会（Open Software Foundation）的一些委员会，他们试图说服用户真正重要的是“开放接口”，而不是操作系统的哲学。这两种主流Unix社区推出的实

施版本大多成了超级臃肿和缓慢的系统。

作为Unix哲学的下一个重要实现，而且是在一个几乎没有什么系统实施该哲学理念的领域，Linux展现出了许多“第三个系统”的特点。有些人厌倦了Unix世界中的自鸣得意和争吵不休，他们在Linux上面找到了很多乐趣。Unix中最好的原创概念在Linux下得以继续发展。许多Linux开发人员都抽出时间，帮助这个系统实现恰当的内核结构、接口和图形用户界面（GUI）。请大家不要忽略一个显而易见的事实：Linux其实就是Unix，只是换了个名字而已。

Linux自身也体现出一些“第二个系统”的特性。比如说，广为人知的“开源软件”系统其实是从Linux世界开始起步的。这个系统要经过一个漫长过程，在人们辩论、举办研讨会、并阐述其优缺点之后，才会进入“第三个系统”阶段。自此，永久自由分发该系统软件的源代码将成为普遍的做法。

如果说Ken Thompson是Unix的创造者，那么Linus Torvalds就是Linux操作系统的发明人，当时他还是芬兰赫尔辛基大学的一名学生。1991年8月25日他发出了那篇现在广为人知的新闻组主题文章，这篇以“嗨，大家好……我正在编写一个（免费）的操作系统”开头的文章对他的命运产生了深远影响。

Thompson 和Torvalds两人至少有一点相似之处，那就是对事物的好奇之心。我们可以找到证据，Thompson编写Space Travel程序只是为了好玩而已。而Torvalds在痴迷于类Unix操作系统——Minix的同时，也完全是因为非常感兴趣才会将流行的Unix命令解释器bash进行改编并运行在他的“玩具”操作系统上。同时，这些在一开始只是“为了好玩”的举动，却最终对软件产业产生了深远影响。

一开始，Linux也不是一款具备可移植性的操作系统。Torvalds无意将它移植到英特尔386之外的其他架构之上。从某种意义上说，他也只是背水一战，因为他的手头只有少量计算机硬件可供选择。因此，最初他并没有采取任何进一步的举措而只是将自己拥有的资源发挥到极致。但是他发现良好的设计原则和扎实的开发模式还是引领着他去把Linux变成一个可移植的系统。从那一刻开始，别的人接过了这个接力棒，很快便将Linux移植到了其他架构。

在Torvalds的Linux出现之前，借鉴他人编写的软件已成为相当普遍的做法。事实上也就是因为这样，Richard M. Stallman才会在具有里程碑意义的GNU公共授权协议（GPL）下正式确立了这一思想。GPL是一个适用于软件的法律协议，基本保证了软件的源代码可以自由提供给任何想要得到它的人。Torvalds最终为Linux采用了 GPL协议，这个举动免

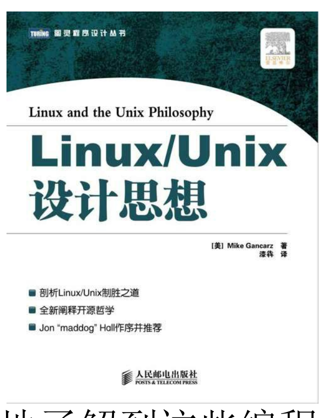
除了所有人对于相关法律与版权纠纷的后顾之忧，让他们可以自由借用Linux的源代码 。由于Torvalds将Linux免费开放出来，因此其他人自然也会将他们的软件免费提供出来以共同发展Linux。

从一开始，Linux已经表现出它确实是一个与Unix非常相似的操作系统。它的开发人员全盘接受了Unix的哲学原理，然后再从头编写了这个新的操作系统。问题是在Linux的世界里，几乎再没有其他程序是重新编写的。一切应用都是建立在其他人写好的代码和概念之上。因此很自然地，Linux成为了Unix系统演变的下一步，或许更准确地说，它是Unix的一个大飞跃。

类似于Unix，在Linux技术发展的早期，有许多开发者参与其中并提供了帮助。不同的是，Unix开发者数量最多的时候也就几千人，而今天Linux的开发者数量却早已达到了几百万之多。这才是登峰造极的Unix！正是这种大规模的开发格局，保证了Unix的后代Linux将在很长时间内都是一款具备强大竞争力的系统。

Linux为Unix世界重新激起波澜，所谓的“开源”要比“专有”软件或是那些没有现成源代码的软件优越。多年以来，Unix开发人员一直坚信这一点。但计算机行业的其他人却被一些专有软件公司的大量宣传所蒙蔽，他们误认为任何借来的或是免费的软件在性能上都无法比拟那些要付费（有时甚至是耗费巨资）的软件。

在市场营销方面，Linux社区也更为精明，他们知道只要市场工作做得好，就算是劣质软件也可以成功销售出数百万份。当然，这并不是说Linux是伪劣产品。只是，有别于它的前身Unix社区，Linux社区认识到，即使是世界上最好的软件，也只有当人们对它产生了解并认识到它的真正价值时，才会为人所用。



开源除了可以让你清楚地了解到这些编程大师们创建系统的方式，还可以激励你去创建更快、更强大的系统。到目前为止，没有一本书同

时介绍 Unix 和 Linux 的设计理念，《Linux/Unix设计思想》将这两者有效地结合起来，保留了 The Unix Philosophy 中 Unix 方面的内容的同时，探讨了 Linux 和开源领域的新思想。本文选自《Linux/Unix设计思想》。

分布式开发样板——**Linux**内核开发过程



作者/ Wolfgang Mauerer

Wolfgang Mauerer资深Linux专家，有数十年Linux开发经验。从1997年最初发表关于内核的系列文章开始，他就醉心于解释Linux核心的内部机制、编写相关的文档。此外，他还著有LaTeX排版方面的图书，其撰写的大量文章已经被翻译成 7 种语言。

显然，算法、数据结构、代码构成了Linux开发的最核心部分，内核就是这些内容。但Linux还有另一个侧面，不应该被忽视：开发了内核的社区、其工作方式、人与人之间交互的方式。这个方面是很有趣的，因为内核是现存最大、最复杂的开源项目之一，它对于大规模的分布式分散开发来说，是一个样板。本文将对内核开发涉及的技术和社会方面提供一个概述。

简介

内核源代码（在主要的README文件中）将开发社区描述为一个“在网络上松散组织的黑客团队”，尽管从开始到现在，内核开发涉及的人数及其职业来源一直在发生变化，但所述说法一直是真实的。这导致的一个直接结果就是开放性：开发者之间大多数的通信都发生在邮件列表上，任何对操作系统演变方式感兴趣的人都可以阅读这些。一个要点在于，来自许多在各方面激烈竞争（请注意，是公司，不是开发者）的公司的开发者在内核开发中密切协作。而非技术人员，通常只能惊讶地站在一边。实际上，这是一项非凡的壮举！

现在，已经不需要多说Linux内核开发的基本原理了。尽管仅仅在15年前创建一个可以实际使用的开源操作系统似乎还是一个妄想，但大多数技术人员已经习以为常。Linux内核开发与经典的开发模型相比，一个本质区别是，前者没有什么固定的形式化规则来规范开发过程如何运作。确实有惯例存在，但很少以文档方式形式化。没有开发路线图，也没有中央代码存储库。但确实有重要的代码存储库和重要的开发者。与

固定的刚性结构相比，这在许多情况下可能都是优点，因为开发过程变得更为动态和灵活。但如果对领域不熟悉，工作也会变得更困难。

内核代码树和开发的结构

Linux内核是一个非常动态的软件，最令人惊讶之处在于，其开发根本没有开发版本！至少没有由Linus Torvalds管理的显式、长期的开发版本。

此前的情况有所不同。传统上，内核开发分为两个不同的分支。一个分支包含稳定的内核版本，应该用于生产系统，其次版本号为偶数。内核分支2.0、2.2、2.4都是稳定分支（2.0.x、2.2.x、2.4.x是发布的各系列版本），而2.1、2.3、2.5是开发版本，诸如2.5.x等。这种方法的基本思想在于，使新的特性和试验性的补丁经历大量的测试和改进，一旦添加了足够数量的新特性，而且代码在实际上以可察觉的方式稳定下来时，就打开一个新的稳定的源代码树。理想情况下，发布商可以从稳定的版本分支取得内核，将其集成到发行版中。

遗憾的是，这种方式运作得不十分好。在打开新的稳定版本之前，一个开发周期可能需要数年，在IT界这是一个非常长的时间了。在新硬件出现时，买家通常不会花几年的时间等待内核的支持（至少是大多数人使用的稳定版本的内核）。不仅对设备驱动程序是这样，对大多数新特性来说，都是如此。因而，发布商确实从开发版内核向稳定分支移植了一些新特性。而且由于每个发行版的“口味”不同，后向移植所选择的特性也不同，这导致了发行版内核之间的分歧越来越大。

从内核版本2.6系列以来，采用了一种新的开发策略。只有一个内核系列，不再划分稳定代码树和开发代码树。而采用了若干比较试验性的内核代码树来测试新的特性，在经过稳定和测试期之后，新特性将直接合并到内核的主系列中。2.6版本的内核代码树由Linus Torvalds管理，读者可能听说过，他是Linux最初的创造者和发起者。来自该代码树的内核通常称之为vanilla内核，以区别发行版根据具体需要修改而来的内核，或各种试验性的代码树。该内核系列通常称之为主线内核。

主线内核代码树之外的代码树，通常在版本号之后增加一个后缀进行标识。主线内核之外，最重要的代码树是`2.6-mm`，由Andrew Morton管理，大多数补丁在被主线2.6内核接受之前，都会先经过该代码树。还存在许多其他子系统相关的代码树，它们通常关注内核的某个特定方面：`2.6-net`关注网络，而`2.6-rt`包含了与实时问题和交互性问题相关的工作，这只是其中两个例子。还有一个`-stable`版本，用于在正式的内核版本发布后，将重要的bug修复集成进来。内核代码树的变动可能

因种种原因而发生：开发者可能失去维护代码的兴趣，如果代码树涉及的问题已经以某种方法解决，那么代码树本身可能也会消失。

命令链

内核的所有活动组件都有一个维护者，他会关注相关的特定领域的开发。许多维护者（特别是大组件的维护者）都被各个Linux厂商雇佣，但有一些仍然是在空闲时间工作。维护者的职责变动范围很大，可能只是控制单个设备驱动程序，也可能涉及一项基础设施（如内核对象机制），更大的范围可能涉及整个子系统(如所有的网络代码、块层或特定体系结构在arch/下的所有代码)。维护者在内核代码树顶层的MAINTAINERS文件中列出，其中包括几项信息，读者在这里可以看到：

MAINTAINERS

```
IA64 (Itanium) PLATFORM
P: Tony Luck
M: tony.luck@intel.com
L: linux-ia64@vger.kernel.org
W: http://www.ia64-linux.org/
T: git kernel.org:/pub/scm/linux/kernel/git/aegl/linux-2.6.git
S: Maintained
```

除了维护者的名字及其电子邮件联络方式，该文件提供了一个邮件列表，供讨论相应领域的开发使用。通常，与直接联系维护者相比，在邮件列表上提出和讨论问题更受欢迎。如果代码通过一个公众可访问的版本控制存储库进行管理，那么文件中会指定存储库的位置，在上面的例子中是一个git存储库，这是许多内核开发者的首选源代码管理系统（git在附录B中讨论过）。最后，还可以指定一个Web页面，其中包含了有关该子系统及其维护状态的信息。原则上，每个信息项都可以用状态Supported和Maintained来区分维护者是否受雇于Linux厂商，但这通常是一个哲学问题。更重要的区别是，受到活跃维护的部分、没有维护者的部分（Orphan）、旧的代码和废弃的代码（Obsolete）、很少被关注但并非完全没有维护的部分（Odd Fixes）。

对内核的各个部分设置维护者，从很小的部分(如驱动程序)到比较大的部分（如整个子系统），这在开发者之间建立了一个松散的层次结构。但没有一个形式上的权威机构来确定这个层次结构，它完全取决于贡献代码的人及其彼此间的信任程度。在代码进入内核时，通常（但并非唯一）的方式是自下而上遍历该层次结构。对代码的修正或新特性通

常首先进入到特定于设备或子系统的邮件列表或到达相应的维护者，接下来向较高层的维护者前进，这些维护者将其传递到Andrew Morton的-mm代码树，[1]由此最终可能合并到vanilla内核代码树中。该过程通常称之为上溯合并（merging upstream）。但这只是一种可能性，规则决非是固定的。

[1]为减少因为新代码彼此不兼容而造成的-mm代码树中合并冲突的数目，在新代码进入-mm树之前，应该用另一个开发系列的代码树-next将此类问题整理出来。

开发周期

放弃明确的开发版内核系列，最重要的理由之一就是希望加速新特性应用到产品版本内核的速度，而不必由发行版厂商来后向移植。该目标显然已经达到：2.6系列中，内核版本发布的间隔大约是70~110天，这意味着每两三个月，就有一个新内核出现。开发工作的许多方面已经由Linux基金会出版的一项研究阐明（[KHCM]），对该研究的更新不时出现在www.lwn.net上。该研究的一个特别有趣的预测是，vanilla内核树的进展是以一种猝发形式进行的，这是故意的。该代码树会等待一些特性就位后，突然向前跨出一步。参见图F-1，该图说明了对内核的修改随时间的变化。

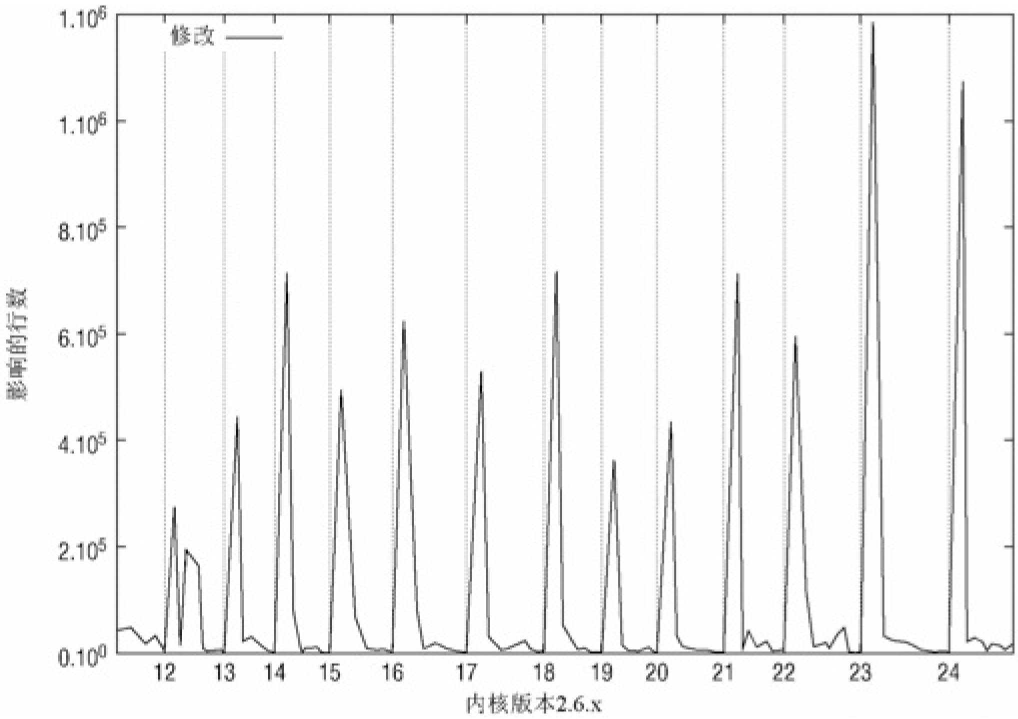


图**1　vanilla**内核树的的变化率。每当打开合并窗口，都会出现一次猝发的大量变更，而后是一段稳定时期，只有相对少的修改

在一个新的内核版本发布后，Linus Torvalds都会打开一个合并窗口（merge window），在比较短的一段时间内保持开放，大约两个星期。

新代码通常只能在这段时间内进入。尽管该规则有例外情况，但该策略的实施是相当严格的。在此期间，代码的变化率是相当大的。在合并窗口关闭时，这段活动时间就结束了，而候选的发布版内核也准备好了。候选发布版提供了一个机会，可以测试各项修改之间的交互，以及识别并修复bug。这段时间的变化率会飞速下降，因为修复通常是非常短的补丁，其重要性等同于初始的特性提交。在一切都稳定以后，一个新的内核版本就发布了。这种行为模式的细节如图2所示，其中给出了内核版本2.6.21到2.6.24的开发进展情况。请注意，*y*坐标轴采用的是对数坐标。虽然第一个候选发布版包含了1 000 000个修改，但下一个发布版这个数字就下降到大约10 000个，后续就降低的更多，直至最后正式发布。

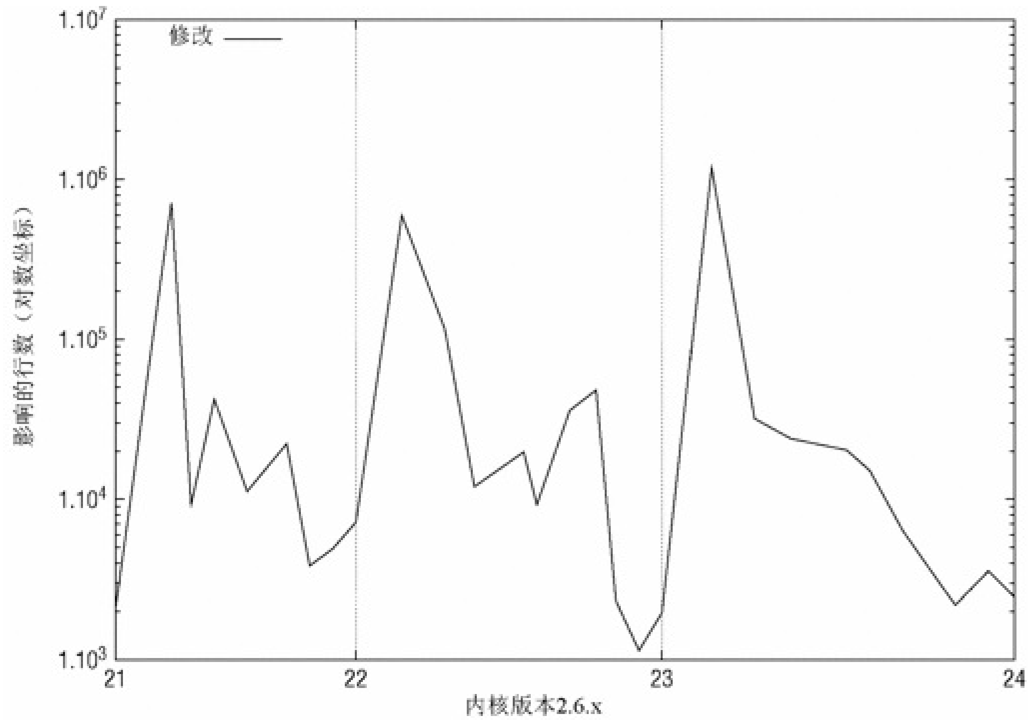


图**2**　对内核版本**2.6.22**到**2.6.24**之间代码变化率的详细分析。请注意，***y***轴采用的是对数坐标

图3的视角稍有不同，该图通过对各个内核版本及候选发布版累积作出的修改来考察代码树稳定的过程。从斜率的剧烈变化可以很清楚地看到合并窗口，而后是比较平坦的曲线，表明这是代码树的稳定期。[2]

[2]这种呈现数据的方式，是受Jonathan Corbet的Kernel Report演讲的鼓舞，在许多Linux相关的会议和类似的场合，都可以看到他的演讲。一些会议的网站(如linux.conf.au)提供了其演讲的视频。

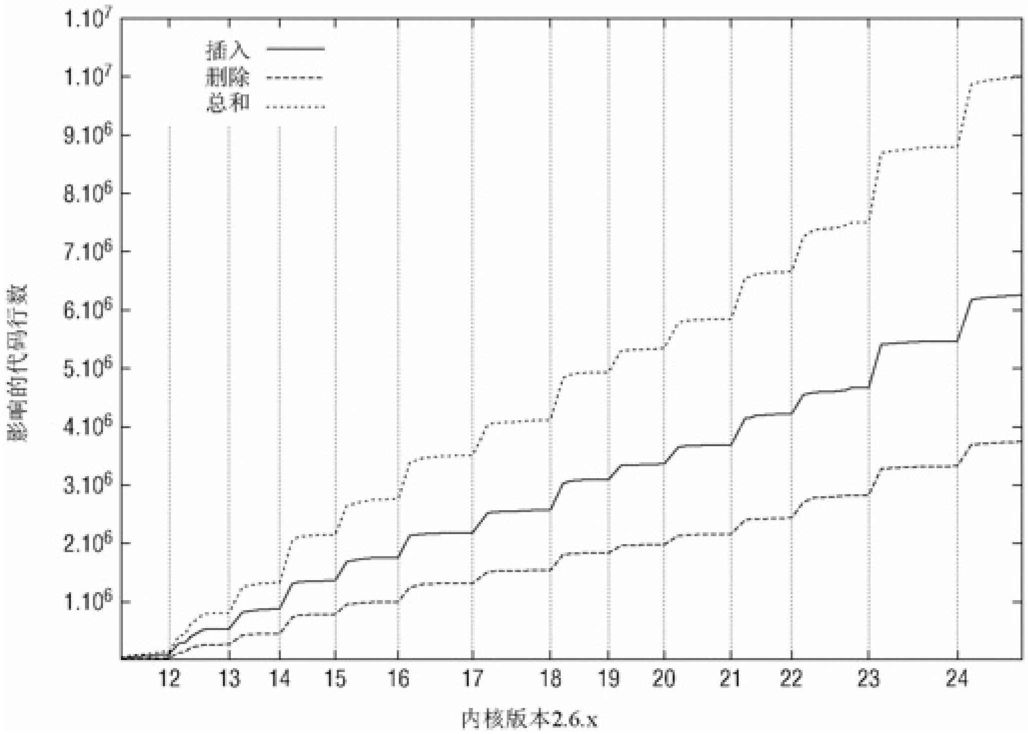


图**3**　在**Linux**开发过程中累积的修改。合并窗口和稳定期的效应看上去非常明显

注意，如果用纯粹的数字来度量软件项目的生产率，总是很困难，特别是这些数字只是基于代码行的增删时。例如，先引进大量代码，然后再删除，合并的效果是没什么意义的，当然按上述方法测量时，将导致很高的变化率。不过呢，这里介绍的这种相对简单的方法可以使读者对开发过程的组织方式获得一个很好的直观理解。也应注意，这样的结果可以非常容易地得到，因为在git存储库中可以获得完整的内核开发历史，读者用类似的方法来自行分析内核源代码中感兴趣的领域，也应该并不困难。

新特性不会突然从天上掉下来，在被主线内核接受之前，它们通常有一段很长的开发历史。开发在这一阶段如何进行，很大程度上取决于特定的子系统和相关的维护者。在代码被主线内核接受之前，可能已经讨论了多年，这并非罕见。例如，Reiser文件系统就花费了很长时间来解决许多开发者对其提出的问题。有时候需要花费大量时间才能将代码推进到vanilla内核中，但有时候进展会快速得多。

例如，Ext4文件系统的开发过程是与vanilla内核密切集成的，其代码从最初开始开发到最终接受广泛测试的整个过程中，一直都处于主线内核中。实际上，其代码库是从Ext3的一个副本开始，然后不断地修改以集成许多新的思想和改进。

在线资源

有大量网站致力于Linux内核开发，它们提供了很多有用信息。因为Web的结构快速变化，在这里进行综述没有实际的意义，因为大多数链接可能都会很快过期。但是，只依赖于所喜爱的搜索引擎来抓取有关内核开发的有用链接，并不是到达成功的最容易路径，特别是需要判断结果的相关性和质量时。因而，下述列表根据作者个人的偏好，提供了一部分较好的链接。

- 当前的内核源代码以及许多基本的用户空间工具都可以从网站www.kernel.org获得。在git.kernel.org上列出了大量的git源代码存储库。
- www.lwn.net是内核开发过程方面的首要信息源，该网站每周定期报道这方面的更新情况。这些更新不仅仅只是内核方面的。该网站收集了Linux开发所有方面的有趣新闻以及IT社区中的相关事件，而优秀的研究文章对各个项目的发展现状给出了深刻的见解。内核在发表一星期之后即免费提供，但大多数最新的信息只对订阅者开放。由于费用较低，我推荐读者尽快订阅！[3]
- 内核的全部修改日志很容易达到若干兆字节。尽管它们细致地记录了到最终版本的每一次提交，但实际上不可能根据日志记录来从一个比较宽泛的角度概览内核开放过程中发生的情况。谢天谢地，www.linuxnewbies.net提供了不那么细致的修改日志，更多地强调全景而不是细节。
- Linux基金会提供了一个“天气预报”服务，试图预测哪些补丁和特性将被未来的内核版本接纳。这项服务是最接近于Linux内核路线图的，它对开发的方向提供了有价值的信息。其URL是www.linux-foundation.org/en/Linux_Weather_Forecast。

[3]当然，我与LWN没有商业利益，也没有任何关系。但这个网站确实令人敬畏。

补丁的结构

内核开发者预期好的补丁应该满足某些固定的条件。尽管这对补丁的准备工作提出了更多要求，但它使得维护者、审阅者、测试者的工作更为容易。因为如果补丁都遵循同样的惯例，就可以减少理解各项修改所需的时间。内核对如何准备补丁包含了详细的指令，可以在`Documentation/SubmittingPatches`中找到。本节将综述要点，但读者在向任何维护者或邮件列表发送代码之前，都务请阅读整个`SubmittingPatches`文档。更多的建议在Andrew Morton的The

Perfect Patch文档中给出，可以在www.zip.com.au/~akpm/linux/patches/stuff/tpp.txt上获得，也可以在linux.yyz.us/patch-format.html上得到。

首要地，必须将大的修改分解为单项修改，与在单个补丁中修改跨越50 000个子目录的1 000万个文件相比，单项修改易于提取要点。一个补丁应该对源代码进行一项逻辑上的修改，即使这意味着需要一个补丁系列，其中有多个补丁会修改同一个文件，也是如此。理想地，补丁应该是可叠加的，即补丁应该是可以独立应用的。但由于修改的性质所致，这不见得总是可能的，在不可能的情况下，应该对应用补丁的正确顺序给出文档。

原则上，补丁序列可以用附录B所述的`diff`和`patch`手工创建。时间稍长，这可能就变得相当乏味，但http://savannah.nongnu.org/projects/quilt的`quilt`工具箱可以使管理补丁栈的大部分工作自动化，从而在一定程度上减轻了工作。

技术问题

就补丁格式的技术方面来说，请注意，一个统一的（unified）补丁要求包含所修改的C函数相关的信息。这样的补丁可使用`diff -up`生成。如果补丁添加了新文件或关注多个子目录下的文件，那么必须使用`diff -uprN`来解决。附录B讨论了形成的补丁的外观，及其包含的内容。

1. 编码风格

内核有一些编码风格要求，定义在`Documentation/CodingStyle`。尽管并非所有开发者都同意该文件中每一个要求，但许多开发者对违反编码风格的做法比较敏感。有一个共同的编码风格是件好事情。内核包含了大量代码，如果补丁/文件使用大量不同的规范，那是真正的麻烦。谢天谢地，Linux内核在编码风格方面不像其他项目那样狂热，但对不受欢迎的风格有着明确的意见，读者可以在下述文档片段中看到：

Documentation/CodingStyle

首先，我建议打印一份GNU编码标准，但不要阅读。烧掉它们，这是一种姿态。

内核开发者预期何种风格呢？要点如下所示。

- 不同缩进层次总是以一个tab分隔，一个tab总是等于8个空格。这对于看过很多用户层代码的程序员来说，可能太大了，但内核是

不同的。很自然，代码在几次缩进之后，会向屏幕右侧快速移动，但这可用作一个报警信号：需要太多缩进层次的代码通常应该替换为更干净的代码，或划分为函数，接下来问题自动地解决了。

比较大的缩进通常会导致字符串和过程参数超过一行80个字符的边界，因而必须明智地分解为块。

除了前述原因之外，许多内核开发者的工作方式有很不寻常的倾向，长时间地专注于代码并非罕见的情况。在连续三天在一行中编写了大量代码之后，视觉可能会变得模糊，大的缩进肯定对这种情况有所帮助（以及大量咖啡饮料）。

- 开始的花括号放在行的结尾，而结束的花括号放在行的开头。在接下来是控制语句时（例如else分支，或do循环中的while条件），接下来的语句不再占用新行，而是接着结束的花括号开始。如果一个块语句中只包含一个语句，那么额外的花括号是不必要的。实际上，不鼓励增加额外的花括号（想一想，从长期来看，这可以为你节省多少次输入）。

函数的惯例不同：开始和结束的花括号都需要独立的一行。
下列代码给出了上述规则的示例：
kernel/sched.c

```
static void __update_rq_clock(struct rq *rq)
{
      u64 prev_raw = rq->prev_clock_raw;
      u64 now = sched_clock();
...
      if (unlikely(delta < 0)) {
              clock++;
              rq->clock_warps++;
      } else {
             /*
              * Catch too large forward jumps too:
              */
              if (unlikely(clock + delta > rq->tick_timestamp +
TICK_NSEC)) {
              if (clock < rq->tick_timestamp + TICK_NSEC)
                      clock = rq->tick_timestamp + TICK_NSEC;
              else
                      clock++;
              rq->clock_overflows++;
              } else {
                      if (unlikely(delta > rq->clock_max_delta))
```

```
                        rq->clock_max_delta = delta;
                clock += delta;
        }
    }

    rq->prev_clock_raw = now;
    rq->clock = clock;
}
```

- 括号内部不应该使用环绕空格，因此if(condition)是反对的，而if(condition)将受到普遍欢迎。关键字(如if)后应接一个空格，而函数定义和函数调用则不需要。前述代码片段也包含了这些规则的示例。
- 常数应该由宏或enum枚举中的成员表示，其名称应该都是大写字母。
- 函数通常不应该长于一个屏幕（即24行）。更长的代码应该分解为多个函数，即使形成的辅助函数只有一个调用者，也是如此。
- 局部变量名称应该简短，不要试图像写小说那样讲故事，比如OnceUponATimeThereWasA-CounterWhichMustBeIntializedWithZero。也可以使用tmp这样的名称，这样也减少了输入次数（同时保护了你的手指）。

  全局标识符应该提供更多有关其自身的信息，因为它们在所有上下文都是可见的。prio_tree_remove是一个全局函数的好名字，而cur和ret则只适用于局部变量名称。由多个表达式注册的名称，应该使用下划线来分隔其组成部分，而不能混用大小写字母。
- typedef被认为是邪恶的化身，因为它们隐藏了一个对象的实际定义，因此通常不应该采用。它可能为补丁的创建者节省一些输入，但将给所有其他开发者的阅读增加困难。

但有时候必须隐藏某个数据类型的定义，例如在需要根据底层体系结构而对一个量提供不同实现的时候，但通用代码不应该注意到这一点。例如，用于原子计数器的atomic_t类型，或页表的各种成员类型（如pte_t、pud_t等）。它们都不能直接访问和修改，只能通过专用的辅助函数，因此其定义对通用代码是不可见的。

所有这些规则（还有更多）都在编码风格文

档Documentation/CodingStyle中进行了讨论，连同其后的基本原理（包括最重要的规则，编号为17：不要重新发明轮子！）。因此，在这里重复该文档中的信息是没有意义的，每一份内核副本都带有该文档，直接去看就可以了！此外，在通读内核源代码时，读者会很快熟悉所要求的风格。

下列两个实用程序，有助于遵守所要求的编码风格：

- Lindent，位于内核的scripts/目录下，它向GNU indent提供命令行选项，以便根据内核首选的缩进选项，来对一个文件重新缩进。
- checkpatch.pl，同样位于内核源代码树的scripts/目录下，它可以扫描补丁文件，以查找违反编码风格之处，并提供适当的诊断。

2. 可移植性

内核可以在大量体系结构上运行，这些体系结构有很多不同之处，对C代码也具有各种不同的限制。新代码的先决条件之一，就是在原则上可能的情况下，该代码应该能够移植到所有支持的体系结构并运行。这里将提醒读者一些重要的、在为内核编写代码时必须考虑的问题。

- 使用适当的锁机制，确保你的代码能够在多处理器环境下安全运行。由于可抢占内核的缘故，这在单处理器系统上同样重要。
- 总是应该编写字节序中立的代码。对小端序和大端序机器，你的代码都应该能够工作。
- 不要假定页长度为4 KiB，而应该使用PAGE_SIZE。
- 不要对任何数据类型假定位宽度。在需要固定数目的比特位时，总是使用具有显式位宽的类型（如u16、s64等）。但读者总是可以假定sizeof(long) == sizeof(void *)。
- 不要使用浮点计算。
- 要记住，栈长度是固定的，有上限。

3. 为代码编写文档

除了为提交的补丁编写文档之外，同样重要的是为代码编写文档，特别是可能从其他子系统或驱动程序调用的函数。内核为此使用了下列特殊形式的C注释：

fs/char_dev.c

```
/**
* register_chrdev() - Register a major number for character devices.
* @major: major device number or 0 for dynamic allocation
* @name: name of this range of devices
* @fops: file operations associated with this devices
*
* If @major == 0 this functions will dynamically allocate a major and
return
* its number.
*
* If @major > 0 this function will attempt to reserve a device with the
given
* major number and will return zero on success.
*
* Returns a -ve errno on failure.
*
* The name of this device has nothing to do with the name of the device in
* /dev. It only helps to keep track of the different owners of devices. If
* your module name has only one type of devices it’s ok to use, for
example, the name
* of the module here.
*
* This function registers a range of 256 minor numbers. The first minor
number
* is 0.
*/
int register_chrdev(unsigned int major, const char *name,
const struct file_operations *fops)
...
```

请注意，注释行以两个星号开始，表明该注释是一个kerneldoc注释。以此类注释开头的函数将包含在API参考手册中，参考手册可以用`make htmldocs`或类似的命令创建。参数名必须以@符号为前缀开始，在生成的输出中将包含对应参数的注释。注释应该包括以下内容：

- 对参数的描述，指定该函数做什么（而不是如何做）；
- 可能的返回代码及其语义；
- 对函数的限制，有效参数的范围，和/或任何必须考虑的特殊问题。

提交和审阅

本节讲述内核开发中两个重要的社会性部分：将补丁提交到邮件列

表，以及后续的审阅过程。

1. 为邮件列表准备补丁

大多数补丁在被考虑包含到任何内核代码树之前，都首先发送到对应子系统的邮件列表，除非你是一个第一流的内核贡献者，能够直接向Linus或Andrew提交补丁。还有一些需要遵守的惯例，如下所述。

- 标题行以[PATCH]开始，标题的其余部分应该对补丁所做的事情作一个简明的描述。一个好的标题是非常重要的，因为它不仅仅用于邮件列表上，在被接受的情况下，它还会出现在git的修改日志中。
- 如果一个补丁不应该直接应用，或需要更多讨论，它可以用一个其他的标识符进行标记，如[RFC]。
- 较大的改动应该分解为多个补丁，每个补丁完成逻辑上的修改。类似地，每个电子邮件只应该发送一个补丁。每个补丁应该以[PATCH m/N]的形式进行编号，其中m是一个计数器，而N则是补丁的总数。[PATCH 0/N]应该包含有关后续补丁的一个概述。
- 对每个补丁自身的更详细描述应该包含在电子邮件的内容部分。同样，在补丁集成后，该说明文本也不会丢失，它也会进入到git存储库中，用作此次修改的文档。
- 代码本身应该直接呈现在电子邮件中，而不能使用任何形式的base64编码、压缩或其他技巧。附件也不特别受欢迎，首选的方式是直接将代码包含在邮件中。任何应该包含在描述中，而不应该进入到git存储库的文本，都应该从补丁分隔开来，可通过一行上的连续三个破折号表示。

很自然，电子邮件客户端不应该进行自动换行操作。有谣言声称，编译器很难接受随机换行的代码。还有，HTML格式的电子邮件是不合适的，纯属多余。

接下来，给出了一个试验性补丁的标题行。它们遵守了此前讨论的惯例：

```
[PATCH 0/4] [RFC] Verification and debugging of memory initialisation Mel Gorman (Wed Apr 16 2008 - 09:51:19 EST)
    [PATCH 1/4] Add a basic debugging framework for memory initialisation Mel Gorman (Wed Apr 16 2008 - 09:51:32 EST)
    [PATCH 2/4] Verify the page links and memory model Mel Gorman (Wed Apr 16 2008 - 09:51:53 EST)
```

```
    [PATCH 3/4] Print out the zonelists on request for manual verification
Mel Gorman (Wed Apr 16 2008 - 09:52:22 EST)
    [PATCH 4/4] Make defencive checks around PFN values registered for
memory usage Mel Gorman (Wed Apr 16 2008 - 09:52:37 EST)
```

请注意，4个包含实际代码的邮件是作为对第一个介绍性邮件的回复发表的。这使得许多邮件客户端可以将发表的邮件归类，更容易将这些补丁识别为一个实体。

看一下第一个邮件的内容：

```
This patch creates a new file mm/mm_init.c which memory initialisation
should
be moved to over time to avoid further polluting page_alloc.c. This patch
introduces a simple mminit_debug_printk() function and an (undocumented)
mminit_debug_level command-line parameter for setting the level of tracing
and verification that should be done.

Signed-off-by: Mel Gorman <mel@xxxxxxxxx>
---

mm/Makefile | 2 +-
mm/internal.h | 9 +++++++++
mm/mm_init.c | 40 ++++++++++++++++++++++++++++++++++++++++
mm/page_alloc.c | 16 +++++++++------
4 files changed, 60 insertions(+), 7 deletions(-)

(PATCH)
```

在概述代码之后，附加了由`diffstat`产生的`diff`统计信息。这些信息可以快速确定一个补丁引入的修改数目，这是以添加和删除的代码行来衡量的，还包括了这些修改发生的位置。这些统计信息对讨论代码是有用的，但没有必要保存到长期的修改日志（毕竟，该信息可以从补丁生成），因此放置在三个破折号之后。接下来是由`diff`生成的补丁，但这就与我们的讨论不相关了，因此不转载其具体内容。

2. 补丁的来源

描述还包含了一个`signed-off`[4]行，标识了补丁的开发者，并用作法律上有效的声明，表明开发者有权利以开源形式发表该代码，通常由GNU General Public License（GPL）版本2涵盖。

[4] signed-off是未经签署而同意，即非正式同意的意思。——译者注

多个人可以sign off同一个补丁，即使他们并非该代码的直接作者。

这表示签字人已经审阅了该补丁，对该代码非常熟悉，并根据其学识判断，该代码能够像声称的那样工作，不会导致数据破坏，不会使笔记本电脑着火，也不会做其他险恶之事。它还跟踪了补丁最终到达vanilla内核树之前，其穿过开发者层次结构的路径。维护者会大量参与sign off活动，因为他们必须审阅大量并非自己编写、但要加入到相应子系统的代码。

在signed-off行上只接受真名，笔名和假名是不能使用的。形式上，补丁的sign off意味着签字人可以证明以下事实：

Documentation/SubmittingPatches

```
Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:
(a) The contribution was created in whole or in part by me and I have the
right to submit it under the open source license indicated in the file; or

(b) The contribution is based upon previous work that, to the best of my
knowledge, is covered under an appropriate open source license and I have
the right under that license to submit that work with modifications,
whether created in whole or in part by me, under the same open source
license (unless I am permitted to submit under a different license), as
indicated in the file; or

(c) The contribution was provided directly to me by some other person who
certified (a), (b) or (c) and I have not modified it.

(d) I understand and agree that this project and the contribution are
public and that a record of the contribution (including all personal
information I submit with it, including my sign-off) is maintained
indefinitely and may be redistributed consistent with this project or the
open source license(s) involved.
```

补丁的sign off在很晚才引入到内核开发中，“三字母公司”（SCO）声称因种种原因他们将拥有所有内核代码，因而所有Linux用户应该把钱都交给该公司，sign off在本质上是对该公司的说法的一种反应。很自然，一些开发者对该公司的这种说法很是不以为然，包括Linus Torvalds本人：[5]

> [5]顺便说及，在Linux内核邮件列表上指责别人是可卡因瘾君子，并不是非常罕见的事情，邮件列表上的对话有时候还是比较粗鲁的。

```
Some of you may have heard of this crazy company called SCO (aka "Smoking
```

```
Crack Organization") who seem to have a hard time believing that open
source works better than their five engineers do. They've apparently made
a couple of outlandish claims about where our source code comes from,
including claiming to own code that was clearly written by me over a
decade ago.
```

实际上，这个案子现在已经几乎成为历史了，而人们（SCO公司的CEO可能是个例外）普遍确信，即使你可能拥有一个简单的最先适配分配器的版权，这绝不意味着一个完整的UNIX内核。不过，由于`Signed-off-by`标记，现在我们可以确切地标识补丁的开发者。

在标记补丁时，还有两种较弱的形式。

- `Acked-by`意味着一个开发者没有直接涉入该补丁，但在经过一些审阅之后认为它是正确的。

  这并不一定意味着`Acked-by`的开发者已经通读了补丁，只是表明他接触到的部分达到了水准。

  举例来说，如果一个体系结构开发者确认（`Acked-by`）称一个补丁对`arch/xyz`目录下执行的修改看起来都没有问题，但是该补丁还包含了`fs/`下的代码，会破坏以M开头的文件中包含奇数个字符的字符串，这种情况下是不能指责进行确认的开发者的。

  当然，上述情况的可能性很小，因为体系结构维护者都非常专业，将根据该文件的“气味”[6]来检测到补丁的破坏性问题，上例只是用来说明概念。

- `CC`用于表示，一个人至少已经知道该补丁，因此他在理论上意识到该补丁的存在，并有机会发表反对意见。

[6]有经验的开发者有时候能够凭借一些表面迹象判断出质量着的源代码，这种迹象被称为“bad smell”。——译者注

在内核版本2.6.25的开发期间，发生了一次讨论，主题是关于代码审阅的价值，以及如何对审阅者评定信用，讨论者就一个解决方案达成一致，引入了所谓的`Reviewed-by`补丁标记。该标记声称：

Documentation/SubmittingPatches

```
Reviewer's statement of oversight
```

```
By offering my Reviewed-by: tag, I state that:

(a) I have carried out a technical review of this patch to evaluate its
appropriateness and readiness for inclusion into the mainline kernel.

(b) Any problems, concerns, or questions relating to the patch have been
communicated back to the submitter. I am satisfied with the submitter's
response to my comments.

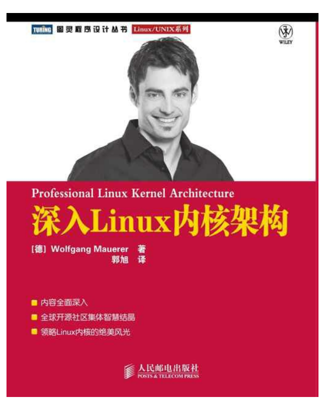
(c) While there may be things that could be improved with this submission,
I believe that it is, at this time, (1) a worthwhile modification to the
kernel, and (2) free of known issues which would argue against its
inclusion.

(d) While I have reviewed the patch and believe it to be sound, I do not
(unless explicitly stated elsewhere) make any warranties or guarantees
that it will achieve its stated purpose or function properly in any given
situation.
```

由于这个原因引入的另一个新标记是Tested-by，读者可以猜测到，它声称补丁已经由签字人测试过，而且在所处计算机上进行的测试是足够的，可以向该补丁增加一个Tested-by标记。

小结

作为实际上最大的开源项目之一，Linux内核之所以有趣，不仅仅是从技术角度来看，而且它也是将开发工作分布到整个世界上和相互竞争的公司的一种新颖而独特的方式。本文讲述了其开发过程是如何组织的，以及对贡献代码有什么要求。在本文中，读者知道了这两者交互的方式，二者之间的不同主要起因于不同的“文化”，以及如何能够最好地弥合二者之间的分歧。



众所周知，Linux操作系统的源代码复杂、文档少，对程序员的要求高，要想看懂这些代码并不是一件容易事。《深入Linux内核架构》结合内核版本2.6.24源代码中最关键的部分，深入讨论Linux内核的概念、结构和实现。具体包括进程管理和调度、虚拟内存、进程间通信、设备驱动程序、虚拟文件系统、网络、时间管理、数据同步等方面的内容。本书引导你阅读内核源代码，熟悉Linux所有的内在工作机理，充分展现Linux系统的魅力。本文选自《深入Linux内核架构》。

Linux的安全问题与安全增强系统

作者/ 赵鑫磊，Jie Zhang

赵鑫磊，阿里巴巴集团资深Linux系统专家。自1998年开始使用Linux，曾参与国内著名的Linux社区发行版MagicLinux的开发工作，是一个土生土长的东北人。工作经历丰富，开过小网吧、大学讲过课、发过小广告（时在阿里妈妈）、干过铁路工（开发铁路车载电视系统）、当过副总裁，但始终没有放弃对Linux的探索与研究。

Jie Zhang（张洁）,阿里巴巴集团支付宝网络技术有限公司高级专家。曾在美国、日本、加拿大和中国的多家高科技企业担任过技术总监等职。参与研发的Jack of Spades Combo卡于1999年3月获得北美权威的PC Magazine杂志的Editors' Choice奖，并赢得PC Computing MVP Finalist Award奖。自1998年开始从事Linux开发，研发基于Linux平台的网络、多媒体、通讯和电子商务等多领域的高科技产品与项目，获中国国家科技进步三等奖，并拥有多项国家发明专利。



Linux的安全问题

计算机安全领域的先驱——Robert Morris 曾经说过："确保计算机安全的三条黄金定律是：不要拥有计算机，不要打开计算机，不要使用计算机。"按照这个说法，世界上就没有安全的计算机。即使是普遍被认为"安全"的 Linux系统，在这方面也存在严重的不足之处。事实就是这样的，不管有多么残酷，您都得照单收着，还得享受着，要不您能咋办？

下面我准备从三个方面来阐述一下 Linux 系统所面临的安全问题，还都挺恐怖的！ Linux可能会说自己很冤，可是没有办法，毕竟这种事儿就发生在你身上了。

黑客入侵

对于个人使用的 Linux系统其实是不必太在意这个问题的。因为你就是那么一小撮儿，而且很可能是被黑客们不太"看得起"的苦逼程序猿。这等于在你身上没啥油水可捞，所以他们是不会打你的主意的。但是作为企业用户就不同，这背后会涉及巨大的商业利益在里面，没有不心动的黑客。而企业又是 Linux 系统应用的大户，所以就成了重灾区。

那么黑客们是如何入侵企业的Linux主机呢？一些类似于密码暴力破解的方法就没必要说了，因为即便现在最傻的系统管理员也知道把密码设置得长一点，所以是基本无用的。当然，那些五花八门的黑客秘籍一类的书上花费 90%篇幅所介绍的什么监听啊、端口扫描啊、等等的一些伎俩，都是骗你银子的，也基本无用。因为现在大家都在使用 ssh 一类的远程控制工具，监听、扫描有什么用呢？我要说的是基于"漏洞"的入侵方式。这种方式一直都是黑客们行之有效，屡试不爽的高招。

要说这些"漏洞"是怎么被黑客们发现的呢？三个字——试，分析。经过不断地尝试和分析，在目前任何一台运行着的 Linux 系统主机上都能找出安全漏洞来。我完全没有夸大的意思，这是事实情况。原因就在于不管什么系统都是人设计和开发出来的，你见过没犯过错误的人吗？"漏洞"就是开发人员所犯下的错误，而且是非常难于被察觉的错误。你还千万别不服气，认为自己十数年的开发工作所积累下来的优良经验，不可能犯下能够产生严重安 全问题的错误。

一个最典型的故事就发生在 Jon Bently的身边，并在他的《编程珠玑》一书中描述过。他说：在多年的时间里先后让上百位专业程序员实现二分查找法，而且每次都是在给出算法的基础描述后，很慷慨地让他们用两个小时的时间去实现它，而且允许他们使用自己所擅长的高级语言（包括伪代码）。令人惊讶地是，大约只有 10%的专业程序员正确地实现了二分查找法。Donald Knuth 也说过：尽管第一次二分查找算法早在 1946 年就被发表，但第一个没有 Bug 的二分查找算法却是十二年后

才被发表出来。 可以说，如果一个简单的二分查找算法没有被正确实现的话，就很有可能产生一个严重危及安全的漏洞。不信，那就先看一下代码 1，我用 C 来实现。

代码 1：

```
static int binary_search( int target, int array[], int size )
{
 int l = 0, r = size -1;
 int h = 0;
 if( size <= 0 )
 return -1;
 while( l <= r ) {
  h = ( l + r ) / 2;
  if ( target > array[h] )
  l = h + 1;
  else if( target < array[h] )
   r = h - 1;
  else
   return target;
        }
 return -1;
}
```

这段代码看起来非常严谨，但是却有着非常严重的问题，你看出来了吗？最关键的是：

```
h = ( l + r ) / 2;
```

因为 h 是int类型，在 32位或 64 位的机器上，它最大能表示的数是 2147483647，如果l 与 r 的和大于这个数会出现什么问题呢？加法溢出，会得到一个负数。如果用一个负数作为数组的下标会出现啥情况？有人说：访问越界，系统崩溃。要我说这还是很优美的方式呢。如果传递给它的数组是在堆中分配①的，系统就会崩溃；但是如果是在栈中分配的，你的堆 栈中的内容就可能泄露了，虽然大多是情况下是无关紧要的，但谁敢保证“有心人”的努力会白费呢？正确的写法应该是：

```
h = l + ( ( r - l ) / 2 );
```

如果使用 Java 语言，可以是：

```
h = ( l + r ) >>> 1;
```

当然，很多人会认为这种错误即便发生了，也不会造成太大损失，毕竟黑客要想从这个漏洞中获取点什么有用的信息甚至入侵到系统中，还是需要付出很多时间代价去剖析或者拥有极大的运气。但是不要忘记，大多数能够被黑客有效利用并成功入侵系统的漏洞，基本上都是这种看似“无关紧要”的错误引发的，比如缓冲区溢出、SQL 代码注入等。而且，你不能认为黑客们就没有中 500 万的运气，也不能低估他们因为利益的驱动而产生耐心的程度。

我给出这个例子要说明的是，很多严重的安全漏洞并不是因为你的粗心大意带来的。因为像上述代码这样简单的例子都要全体计算机工作者们经过 12 年的实践才能发现。而作为Linux 这样一个复杂的软件系统，又有哪些隐患至今还是不为人知的呢？而且一个很实际的问题就是，随着 Linux系统的广泛应用，其所暴露出来的安全漏洞也越来越多了。

“病毒泛滥”

如果严格说 Linux 病毒泛滥是很冤枉的，只是我们将病毒、木马、流氓软件等统统称为了病毒罢了。在我看来，它们不但是计算机病毒，而且还是这个社会的毒瘤。因为现在的病毒制造者根本就不是什么喜欢恶作剧的电脑神童，其真实的本质是趋炎附势的财奴。不但危害着企业的利益，还侵害着我们这些兢兢业业、艰苦奋斗的房奴。

就 Linux 本身而言，从科学层面定义的计算机病毒是很难在 Linux 系统中长期生存下去的。从历史上看，1996 年秋诞生了 Linux 上的第一个病毒 Staog，后来在 1997 年初又诞生了第二个病毒 Bliss，它们两者之所以能够成为病毒是因为当时 Linux 内核存在某种缺陷，而当缺陷被修正了之后，它们就随之消亡了。直到现在，十几年过去了，没有发生任何实质性的Linux 病毒大流行的情况出现，更谈不上“泛滥成灾”。可以说 Linux 的病毒基本就是人们茶余饭后的谈资罢了，因为一个成功的Linux病毒必须是手法高超且具有非凡创意的软件作品，这留给真正的天才们去创造吧，那些财奴们才懒得做这样的事情呢。

但是 Linux上的恶意软件却有愈发疯狂的趋势。虽然 Linux是以开放源代码著称，但是普通用户有几个能有那么大把的时间和能力，每个软件都从源代码安装，且能够对每一个常用软件的代码都了如指掌呢？这

就给某些财奴们创造了机会，而且有时候还能冒充正义的化身，高喊着支持开源软件运动，而在背后却实行着他们不可告人的勾当。使得很多无辜的开发者成为他们替罪的羔羊，很多善良的人们沦为他们赚取大把钞票的工具。随着Android这样的基于 Linux 平台的移动操作系统的普及，以及 Google 等这样极度崇尚所谓“美式自由”的公司允许人们任意开设应用商店的境况出现，使得这种现象开始有愈演愈烈的趋势。

可是，很多问题并不是财奴们单方面就能掀起这么大风浪的。毕竟中国的一句古话是：“一个巴掌拍不响”。很多 Linux 用户也有自身的问题。总是不爱受到限制，喜欢使用 root 账号，这使得病毒、木马、流氓软件有了在Linux 下生根的更广阔的空间和养分。我仅能在这本文中卑微地呼吁一下：亲们，善待 Linux，慎用 root 账号，保护你自身的权益！

拒绝服务攻击

拒绝服务攻击也叫 DoS（Denial Of Service）攻击，这跟曾经十分流行的操作系统没有半毛钱关系。这是一种非常缺德地侵犯计算机系统安全的行为。虽然遭到这种攻击的用户不会爆出什么“艳照门”来引爆大众的眼球，但损失依然会很惨重。之所以说 DoS 很缺德，是因为它如果针对你个人，你将无法正常使用你的计算机，除非断开网络，但是没有网络我们还怎么活啊？要是它攻击某些商业性服务器，那么不单提供服务的系统会瘫痪，所有需要得到服务的用户也将失去服务。然而实行这种攻击的“人”却得不到什么实际的好处。我之所以将人字加上引号，就是因为这种“损人不利己”的行为应该已经超出人类的价值观了，虽然有些情况是出于某种所谓“商业”目的。 其实以现在个人计算机的运算能力，加之 Linux 系统优秀的 TCP/IP 协议栈实现，普通的 DoS 攻击并不会带来什么特别大的威胁。面对强大的商业服务器系统，DoS 自然也不在话下了。这是 Linux 的一种优势。但是，当前“黑客”（依然要用引号）们已经变换了策略，采取了更为恐怖的 DDoS（Distributed Denial of Service），翻译过来是分布式拒绝服务攻击。

这种攻击的危害相当可怕，让很多 DoS 攻击源同时向某个计算机发起攻击，犹如洪水猛兽一般势不可挡，而且甚至与被攻击计算机处于同一网络的其他无辜计算机也会受累。国内最为著名的发生于 2009 年5月19 日的“暴风门”事件就是一起规模空前的 DDoS 攻击，直接导致了中国电信的大面积网络瘫痪，面积覆盖大半个中国。足可见 DDoS 的可怕之处。

面对这种类型的攻击，可以说任何操作系统都是无能为力的。因为这涉及了一个计算机科学历史性的“巨大 Bug”—— TCP协议的机制问

题。TCP协议是一种面向连接的协议。两台计算机通过 TCP 协议通信时，这两台主机要负责维护这个连接。这必然会消耗一定的计算资源和操作系统资源，比如文件描述符。作为服务端的计算机往往需要同时维护多个 TCP 连接，但是这个数量是有限制的，因为文件描述符会占用内存，纵使可以放开描述符的限制也会遇到内存上的限制，这个你没法放开。DoS 或 DDoS攻击就利用了这个“漏洞”，把计算机系统的全部资源都消耗干净而让它无法继续工作。当然，这还是很普通的手法。更厉害的还有 TCP Sync Flood、UDP Flood 等，本文就不一一介绍了。总之是防不胜防。

虽然DoS或DDoS 从原理上讲是目前互联网技术无法解决的，但是 Linux 的开发者们，包括其他操作系统的开发者们，依然在试图改进一些 TCP/IP 协议的实现算法，来提高系统本身对抗这种攻击的能力。根据我的经验，Linux 在这方面不如 Windows做得好。在我的从业经验中，Windows 2003 的服务器在遇到这类攻击时的存活时间往往比 Linux 服务器要长。 而且有些版本的 Linux内核在遇到这种攻击后，会出现“Kernel panic”，整个系统彻底崩溃。Windows 2003 却从来没遇到过。换句话说，在攻击结束之后，Linux 服务器需要重启才能继续工作，而 Windows 2003 服务器能够立即开始工作。这就相当于 Linux系统在遇到类似问题时，往往需要更长的系统修复时间，使得整套系统的可靠性下降。虽然 Linux有免费的优 势，但是在遇到这种问题的时候，对于某些用户来说，Windows 2003 会更划算。

虽然我在介绍 Linux，但是在这里却一直赞扬 Windows 很强。并不是我有什么叛逆的心态，只是不爱护短罢了。同时也是想让大家了解 Linux 在安全方面相较于其他系统还是有不足的地方的，不要去迷信。

安全增强系统

在大多数用户的眼里，每次登录 Linux 系统都必须输入密码，而且除了 root 之外，一个普通用户不能随意读写其他用户的文件，也不能更改系统设置，应该足够安全了。当了解到PAM那样细致入微的用户认证机制之后，更加坚定了对 Linux 系统安全性的信心，似乎从此以后就可以高枕无忧了。且慢，有关安全性的讨论我们才刚刚开始，危机依旧四伏，而且问题很多时候就恰恰出现在 PAM 那看似细致入微的机制中。

什么？PAM 机制不安全？并不是，只是因为 PAM 是用户认证机制，而不是用户授权机制。真正的用户授权机制在 Linux的内核，而且这套机制存在问题。

主动访问控制**DAC**

Linux 内核的用户授权机制也叫访问控制机制，用户身份问题和文件

管理方式就是受控于这种机制。Linux 系统中的用户主要分为 root（系统管理员）和普通用户两种，并将他/她们划分到不同的群组中。而这两种用户是否能够访问系统中的某个文件则与该文件的 rwx 权限属性有关。如果某个程序想要访问这个文件，Linux内核会根据该程序的拥有者的UID和所属群组的GID与该文件UID和GID的rwx权限属性进行对 比来决定是否允许操作。

Linux 这种控制文件访问的机制有一个非常好听的名字——主动访问控制，英文的叫法是 Discretionary Access Control，简称DAC。名字虽然好听，但是缺陷可一点都不少：

- root 拥有特权

  特权这种事儿，在任何时候都是十分可怕的。root 在 DAC 中不受任何限制，如果某个拥有 root 权限的程序被一小撮别有用心的人所控制，那么他就可以利用这个程序在你的系统上干任何事情，而且你还不知道。
- SUID程序的权限升级

  Linux 为了实现类似让用户能够修改自己密码的操作（影子文件只能由 root 存取），只能给某些程序开启一个后门，这个后门就是SUID。这可是特权的直接产物，因为有特权就会有人通过特权走捷径。虽然设计良好的带有 SUID的程序都会集成PAM，但是万一在实现上有什么闪失的话，依然会被一小撮别有用心的人所利用，并且很容易就能将权限提升到 root，那么他就可以利用这样的程序在你的系统上干任何事情，而且你还是什么都不知道。
- 用户可以利用程序来更改文件的存取权限

  如果某个年轻的 Linux 用户为了自己的方便，将某个目录的权限设定为 777，那么该目录就可以被所有人任意访问。这可是一件非常危险的事情，别人想在这个目录下干什么就干什么，你依然是什么都不知道。

其实缺陷还有很多，但是已经无需多举了，这已经足够让所有人倒吸一口凉气了。而且对于这些缺陷更为要命的是，防火墙、入侵监测系统等完全无能为力，是不是该擦擦背后的冷汗了呢？

那么 Linux 是否在面对安全问题时就完全缴械投降了呢？没有，因为还有 MAC。千万不要以为这是“乔帮主”的发明，也跟网卡的物理地址无关，它是……

强制访问控制**MAC**

与主动访问控制相对应的是强制访问控制，英文的叫法是 Mandatory Access Control，简称 MAC。

MAC 可以规避 DAC 的所有问题。因为 MAC 针对特定的程序与特定的文件进行关联性地访问控制。即便你是 root，在使用不同的程序的时候，你所能访问的文件也只是那个程序所能访问的文件。换句话说，MAC 控制的主体是程序而不是使用者，MAC 规定了一个程序能够访问哪些文件，而跟使用这个程序的用户无关。

但是，一个 Linux 系统中的程序数量比它的用户数量多个几十倍是一点都不奇怪的，那么每个程序都使用 MAC 来控制似乎不太现实。Linux 系统采用了一个折中的方案——DAC与 MAC 结合使用。

对于大多数与安全无关的程序就使用 DAC 来控制好了。对于那些安全攸关的程序，比如 passwd、httpd 等则采用 MAC 来控制，只能让它们读取/etc/shadow 文件或访问/var/www目录，即便它们存在某些问题能够被那些别有用心的人所利用，又能掀起多大的风浪呢？而且即便是使用 MAC 控制的程序，也首先要进行一次 DAC 的控制。这样，整个系统的访问控制能力不但细腻得多，安全得多，也容易使用了。

那么 Linux 是怎么实现 MAC 的，我们又该怎么使用呢？不用着急，答案马上揭晓。就像 DAC 那样，MAC 也是由 Linux 内核来实现的，比较经典的有 SELinux、AppArmor 和Grsecurity，其中 SELinux 已经包含在 CentOS / RHEL / Fedora Linux、Debian / Ubuntu、Suse、Slackware 等很多发行版中，我们就以 SELinux 来认识一下访问控制是如何提供强健的安全保证和防御未知攻击的吧。

SELinux——Linux的MAC实现

作为 Linux新老用户如果你说不认识它，那我就会怀疑你之前用的是 Linux 吗？因为这个东西在 Linux 下扎根已经有 10 年（到 2013 年）的时间了，而且所处的位置也是十分显耀的根目录下。但是也不管你是 Linux 新用户还是老用户，不知道 SELinux 是 MAC 也完全不奇怪，甚至老用户们根据过往的经验还会十分讨厌 SELinux。那么从现在起，我就让新用户了解它，老用户喜欢它。

1. SELinux 到底是什么？

首先，SELinux 是一种基于策略的 MAC 安全系统，是以 Linux 内核可加载模块的形式提供的。

其次，SELinux 的全称是 Security Enhanced Linux，翻译过来就是安全强化 Linux，是Linux 上最杰出的新型安全子系统。

另外，SELinux 的来头也着实不小。是由美国国家安全局（NSA，The National Security Agency）和SCC（Secure Computing Corporation）

开发，所以它的能力是不容小觑的。

最后，SELinux 是在 2.6 版本的 Linux 内核中开始被引用。对于目前可用的各种 Linux安全模块来说，SELinux是功能最全面，而且是测试最充分的，是在超过 20 年的 MAC 研究基础上建立起来的。自从 Linux 拥有了 SELinux，大型企业和政府部门才真正地开始考虑采用 Linux 作为其主要的操作系统。可以说 SELinux是 Linux的一个十分显耀的卖点。这也说 明但凡能叫得响的 Linux 发行版，基本上都会集成 SELinux。

2. 开启 **SELinux**

非常遗憾的一件事情是，搜遍 Google 和百度，看到的都是如何关闭 SELinux。或许它真的给你带来了不小的麻烦，但是如果你是一个十分谨慎的人，那么就从现在起，开启SELinux 吧。

但是当我准备提笔的时候，却突然发现不知道该如何下手了。因为不同的 Linux发行版开启 SELinux 的方式有很大不同，而且即便是相同的发行版的不同版本也会有很大变化。这的确是一个非常难办的事情。就冲着这一点，我认为大部分用户都迫切地要关闭 SELinux 并不是盲目的行为，因为它真的很“讨人厌”。但是话分两头说，SELinux 在配置的时候虽然有些烦人，但是安全性那可是杠杠地，从日益恶化的安全局面上看，还真的很难找出不开启SELinux 的理由。

那么所幸我就找一个比较通用的Linux发行版做为例子吧，如果你选择了其他的发行版，那就问问 Google 和百度吧，虽然都是告诉你如何关闭SELinux 的，但是反向操作不就是开启了吗？哦，差点忘记说了，我选择了 CentOS 6.x 作为参考例子。选择它的原因是因为这玩意儿本身就是RHEL，多数企业用的都是它。而且在后面所讲解的所有内容，也都是基于它的。

好了，废话不多说了。打开/etc/sysconfig/selinux这个文件，你可能会看到下面的内容：

```
#This file controls the state of SELinux on the system.
# SELINUX= can take one of these three values:
#     enforcing - SELinux security policy is enforced.
#     permissive - SELinux prints warnings instead of enforcing.
#     disabled - No SELinux policy is loaded.
SELINUX=disabled
# SELINUXTYPE= can take one of these two values:
#     targeted - Targeted processes are protected,
#     mls - Multi Level Security protection.
SELINUXTYPE=targeted
```

　　如果不是这样，可能你并没有关闭 SELINUX，看看是不是跟下面的差不多：

```
# This file controls the state of SELinux on the system.
# SELINUX= can take one of these three values:
#     enforcing - SELinux security policy is enforced.
#     permissive - SELinux prints warnings instead of enforcing.
#     disabled - No SELinux policy is loaded.
SELINUX=enforcing
# SELINUXTYPE= can take one of these two values:
#     targeted - Targeted processes are protected,
#     mls - Multi Level Security protection.
SELINUXTYPE=targeted
```

　　发现什么地方有变化了吗？就是将 SELINUX=disable 变更成 SELINUX=enforcing 了。

　　disable 很好理解，就是关闭嘛！那么 enforcing 对于 SELinux 意味着什么呢？它代表了一种工作模式，强制 SELinux 使用你所定义的所有安全策略。那么与 enforcing 模式相对应的是 permissive 模式。如果 SELinux工作在 permissive 模式之下，则并不代表系统已经真正受到 SELinux 的保护了，因为它只要求显示一些相关的信息状态，余下的就什么都不做了。

　　开启和关闭 SELinux 都要重新启动系统，因为这个需要内核本身被重新加载。如果你不确定 SELinux 是否已经成功开启了，可以使用 setatus 命令来确认，在我的系统下，会得到这样的结果：

```
SELinux status:     enabled
SELinuxfs mount:   /selinux
Current mode:    enforcing
Mode from config file:  enforcing
Policy version:     24
Policy from config file: targeted
```

　　虽然开启或关闭 SELinux 要重启系统，但是在 enforcing 和 permissive 模式之间进行切换就很方便了。只需要使用 setenforce命令就可以，给它一个参数0或1。0就代表 permissive模式，那么 1 就代表 enforcing 模式啦。如果要查看当前 SELinux 工作在什么模式下，可以使用 getenforce 命令来获得。

需要注意的是，一旦你开启了 SELinux，而且设置它工作在 enforcing 模式之下，很可能会有悲剧发生，你有一堆服务无法顺利启动。最主要的提示信息是/lib 目录下的某些文件没有读取权限而导致启动失败。一旦发生这种悲剧，可以将 SELinux 设置为 permissive 模式，然后使用“restorecon -Rv/”命令。如果有人很倒霉或者出于好奇将 SELINUXTYPE=targeted设置成了 SELINUXTPE=mls，那可能会是更大的悲剧，因为你都无法登录系统也无法关机。

如果尝试断电的话，就更加悲剧了，单用户模式你都进不去了。这个时候怎么办呢？动用内核启动参数吧，我们之前介绍过了。在 GRUB 的交互界面中指定内核参数 enforcing=0 就行。启动之后就是喜剧了：）

如果你在读完本节的内容之后，还是觉得 SELinux 有些累赘，不妨将它设置为permissive 模式，这样当你的系统受到安全威胁的时候可以使用 dmesg 命令来查有关信息。当然，这个模式并不是真的让你这么干的，它存在的目的是让你能够调试一下你所设置的策略是否合理。

至于后面那个 SELINUXTYPE=targeted 或者 SELINUXTYPE=mls 是怎么回事，我们后面再做详细探讨。

3. 安全上下文

在进一步讲述 SELinux之前，我们先来了解一下它的最基本的概念——安全环境，或者说是安全上下文（context）。

安全上下文实际上是一个标签，如果系统中开启了 SELinux，那么系统中所有的文件、目录、进程、套接字、乃至用户本身就会带有这个标签。这个标签由：用户、角色、域/类型、敏感度和类别这五个部分构成，并使用冒号“:”分割。这五个部分并不总是全部出现，具体的要根据你选择了何种策略。使用“id -Z”命令可以查看当前用户的安全上下文，在我的系统中是这个样子的：

```
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

要查看进程的安全上下文，可以使用“ps -Z”命令，至于文件或目录的安全上下文，可以使用“ls -Z”命令。

安全上下文中的第一个部分是用户。它与 Linux本身的用户是完全不同的概念。它们可以共存，而且很多时候还会被人们弄混。因为一个 SELinux 的用户会与 Linux 本身的用户使用相同的文本表达（登录名）。但是不管怎样，都要记住它们两个完全不是一回事儿。使用su 命

令并不能改变 SELinux 的用户。

每一个 Linux 本身的用户可以和一个 SELinux 的用户相对应，也就是说当他/她通过合法途径登录了一个 Linux 系统，那么同时就会分配给他/她一个 SELinux 用户。可以通过“semanage login -l”命令来查看这种对应关系，在我的系统中是这个样子：

```
Login Name  SELinux User  MLS/MCS Range
__default__  unconfined_u  s0-s0:c0.c1023
root   unconfined_u  s0-s0:c0.c1023
system_u  system_u   s0-s0:c0.c1023
```

安全上下文中的第二个部分是角色。角色规定了用户能干什么，这跟我们现实社会中的角色很像，比如：有些人是医生、有些人是教师、有些人是官员……一个人如果有了社会角色，那么他/她就会得到某些别的角色无法获得的资源，比如：医生可以给病人下处方、教师可以授予学生学位、官员可以决定你是不是医生或是教师……

每一个 SELinux 用户都会有它对应的角色，而且还可以对应多个角色，现实社会是这样的，比如：一个医生也可以是老师，去培养新一代的医生。

4. 身份

在 SELinux中，身份的概念与传统的 Linux UID是完全不同的，它们可以共存。但是在很多时候会被人们弄混，因为一个 SELinux 的身份会跟标准的 Linux 登录名使用相同的文本表达（大多数情况都是这样），但是不管怎样都要记住，它们两个完全不是一回事儿。在SELinux 中身份是安全上下文的一部分，它会决定什么可以被执行。运行 su 命令是不会改变SELinux的身份的。

5. SELinux 的策略

一开始我们就说 SELinux 是一种基于策略的 MAC安全系统，那么它都提供了什么策略呢？回答这个问题是很难的，因为 SELinux 的策略可以有千百种，我却完全不知道你到底喜欢哪一种。

但是对于 SELINUXTYPE 这个配置项留给我们的选择并不多，因为 Linux 发行商们已经为我们编制好了一些策略套件，拿来直接使用或做一些小幅度的改动就可以放心地使用了。而这个 SELINUXTYPE 配置项就是用于选择策略套件的。可是话又说回来，既然已经编制好了，干嘛还要给我们选择呢？只能说安全这玩意儿，真是众口难调啊！

对于 CentOS 6.x，它提供了三个策略套件，分别是 targeted、mls 和

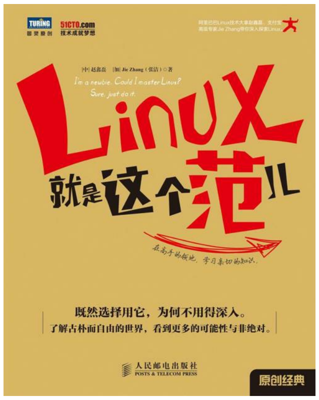
minimum，也分别代表了不同的侧重点。targeted 是最常用的策略套件，也是 CentOS 6.x 默认的。它侧重的是对网络服务的严格限制，用来在没有严重影响用户体验的情况下尽可能多的保护关键进程，在 CentOS 4只有 15 个已经定义好的 targets 存在，包括 httpd、named、dhcpd、和 mysqld，而后来的 CentOS 版本受保护的对象越来越多，targets 数量猛增，现已超过了 200 个。在保护安全的同时 targeted 策略也期望能达到这么一种境界，大多数用户没有感觉到 SELinux 在运行而它却在默默起着保护作用。至于其他没有被 SELinux保护的行为仍旧只能要靠古老稳定的 UNIX 安全系统来保障了。SELinux 还允许用户使用多策略，我们可以使用 mls 套件来实现。它能做到不需要对安全策略进行破坏性地改变就能使 MLS 生效，我们需要在文件/etc/sysconfig/selinux中把SELINUXTYPE=targeted这行改成 SELINUXTYPE=mls，使用chcon或 semanage 命令重新标签上下文，然后重启系统。如果有的用户只想使用一个“小”的SELinux 在内存不多的虚拟机上运行，他可以采用 targeted 策略慢慢删除不需要的软件包，但是这样白手起家地开始工作迟早有一天会让他泄气的。所以针对这种需求，SELinux 推出了和 targeted策略一样的 minimum软件包，先只要求安装 base policy软件包，然后包含所有targeted 策略的 selinux-policy-minimum rpm包进行后安装再加载到内核里。

小结

除了通过用户认证机制和访问控制机制提高 Linux 系统的安全性外，把入侵检测系统（LIDS）集成到 Linux 内核中，可以进一步加强 Linux 内核的安全性。

另外，“亡羊补牢，为时不晚”，sysylogd 被用来处理系统日志。虽然这已经是事后诸葛亮，但是它是系统安全与管理的一个重要方面，帮助我们排查错误原因防止类似的问题再次发生。syslogd 记录的日志一般在/var/log/下，当然也有存储在另外的服务器上的。它可以记录 who、when、where 和 what 等要素，这样你就可以知道系统什么时候重新引导过、软硬 件错误、系统运行的服务信息等。

有时日志还会包含一些敏感数据，例如本地启用了哪些服务，用户账号和配置数据，在这种情况下需要考虑加密存储与传输数据。



《Linux就是这个范儿》通俗易懂、深入浅出，从Linux设计思想的角度出发，授人以渔。同时，又密切结合应用案例，透彻展示出Linux的系统架构设计是如何在实际中贯穿的，从而让读者真正掌握Linux的强大之处。书中不含较为复杂的技术理论问题，一切从实际出。本文选自《Linux就是这个范儿》。

你可能不知道的Shell

作者/ 弘毅
独立开发者，曾就职于微软亚洲研究院。先后毕业于清华大学和Johns Hopkins University。热爱开源软件和极客文化。

Shell也叫做命令行界面，它是***nix**操作系统下用户和计算机的交互界面。**Shell**这个词是指操作系统中提供访问内核服务的程序。

这篇文章向大家介绍**Shell**一些非广为人知、但却实用有趣的知识，权当品尝**shell**主食后的甜点吧。

先科普几个你可能不知道的事实：

1. Shell几乎是和Unix操作系统一起诞生，第一个Unix Shell是肯·汤普逊（Ken Thompson）以Multics上的Shell为模范在1971年改写而成，并命名Thompson sh。即便是后来流行的bash（shell的一种变体），它的年龄实际上比当前流行的所有的Linux kernel都大，可谓在Linux系统上是先有Shell再有Kernel。

2. 当前绝大部分*nix和MacOS操作系统里的默认的Shell都是bash，bash由Brian Fox在1987年创造，全称Bourne Again shell (bash)。

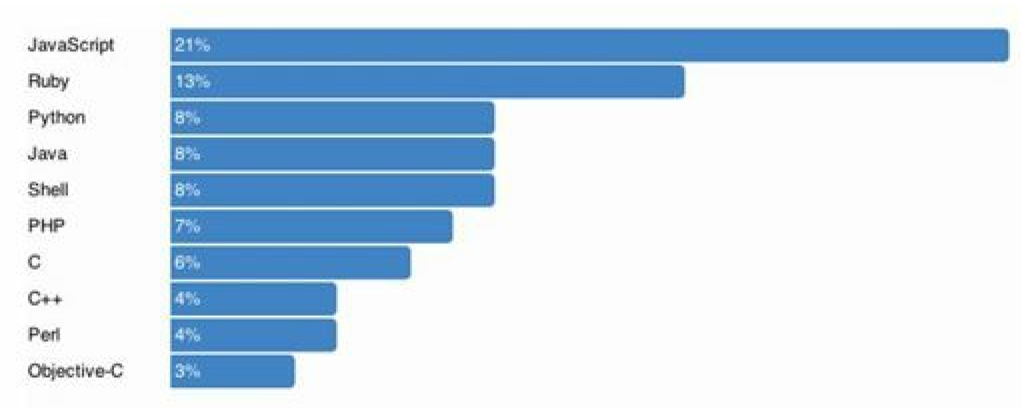
3. 你或许听说除了bash之外，还有Bourne shell (sh)，Korn shell (ksh)，C shell （包括 csh and tcsh），但是你知道这个星球上一共存在着大约50多种不同的shell么？想了解他们，请参考 http://www.freebsd.org/ports/shells.html。

4. 每个月tiobe上都会给一个编程语言的排名，来显示各种语言的流行度。排名指数综合了全球范围内使用该语言的工程师人数、教学的课程数和第三方供应商数。截止至2012年11月份，tiobe公布的编程语言排行榜里，bash的指数是0.56%排名22位。如果算上它旗下的awk 0.21%和tcl 0.146%，大概就能排到14名。注意这里还不包括bash的同源的兄弟姐妹csh、ksh等，算上它们，shell家族有望接近前十。值得一提的是一直以来shell的排名就很稳定，不像某些“暴发户”语言，比如objective-c，这些语言的流行完全是因为当前Apple系的崛起，但这种热潮极有可能来得快去得更快。

Position Nov 2012	Position Nov 2011	Delta in Position	Programming Language	Ratings Nov 2012	Delta Nov 2011	Status
1	2	↑	C	19.224%	+1.90%	A
2	1	↓	Java	17.455%	-0.42%	A
3	6	↑↑↑	Objective-C	10.383%	+4.40%	A
4	3	↓	C++	9.698%	+1.61%	A
5	5	=	PHP	5.732%	-0.36%	A
6	4	↓↓	C#	5.591%	-1.73%	A
7	7	=	(Visual) Basic	5.032%	-0.01%	A
8	8	=	Python	4.062%	+0.45%	A
9	10	↑	Perl	2.182%	+0.10%	A
10	11	↑	Ruby	1.739%	+0.24%	A
11	9	↓↓	JavaScript	1.278%	-1.29%	A
12	16	↑↑↑↑	Delphi/Object Pascal	0.995%	+0.12%	A
13	13	=	Lisp	0.951%	-0.23%	A
14	14	=	Pascal	0.881%	-0.11%	A
15	23	↑↑↑↑↑↑↑↑	Visual Basic .NET	0.769%	+0.24%	A-
16	19	↑↑↑	Ada	0.662%	+0.04%	B
17	12	↓↓↓↓↓	PL/SQL	0.632%	-0.81%	B
18	18	=	Lua	0.631%	0.00%	A-
19	15	↓↓↓↓	MATLAB	0.620%	-0.34%	B
20	24	↑↑↑↑	Assembly	0.585%	+0.06%	B

Position	Programming Language	Ratings
21	SAS	0.577%
22	Bash	0.562%
23	Transact-SQL	0.473%
24	COBOL	0.471%
25	ABAP	0.442%
26	Fortran	0.437%
27	Scheme	0.418%
28	R	0.417%
29	Scratch	0.406%
30	Logo	0.389%
31	Prolog	0.360%
32	Erlang	0.339%
33	RPG (OS/400)	0.330%
34	Scala	0.324%
35	Haskell	0.323%
36	D	0.283%
37	Smalltalk	0.254%
38	APL	0.236%
39	Forth	0.228%
40	ML	0.226%

5. 全球最大的源代码仓库Github里，shell的项目数占到了8%，跻身前5和Java相当，可见在实战工程里，shell可谓宝刀不老。



再分享一些可能你不知道的**shell**用法和脚本，简单**&**强大。

*阅读以下部分前，强烈建议读者打开一个shell实验，这些都不是shell教科书里的大路货哦：）

1. sudo !!

以root的身份执行上一条命令 。

场景举例：比如Ubuntu里用apt-get安装软件包的时候是需要root身份的，我们经常会忘记在apt-get前加sudo。每次不得不加上sudo再重新键入这行命令，这时可以很方便的用sudo !!完事。

2. cd –

回到上一次的目录 。

场景举例：当前目录为/home/a，用cd ../b切换到/home/b。这时可以通过反复执行cd –命令在/home/a和/home/b之间来回方便的切换。

3. ^old^new

替换前一条命令里的部分字符串。

场景：echo "wanderful"，其实是想输出echo "wonderful"。只需要^a^o就行了，对很长的命令的错误拼写有很大的帮助。

4. man ascii

显示ascii码表。

场景：忘记ascii码表的时候还需要google么?尤其在天朝网络如此“顺畅”的情况下，就更麻烦在GWF多应用一次规则了，直接用本地的man ascii吧。

5. ctrl-x e

快速启动你的默认编辑器（由变量$EDITOR设置）。

6. netstat –tlnp

列出本机进程监听的端口号。

7. tail -f /path/to/file.log | sed '/^Finished: SUCCESS$/ q'

当file.log里出现Finished: SUCCESS时候就退出tail，这个命令用于实时监控并过滤log是否出现了某条记录。

8. ssh user@server bash < /path/to/local/script.sh

在远程机器上运行一段脚本。这条命令最大的好处就是不用把脚本拷到远程机器上。

9. screen -d -m -S some_name ping my_router

后台运行一段不终止的程序，并可以随时查看它的状态。-d -m参数启动“分离”模式，-S指定了一个session的标识。可以通过-R命令来重新“挂载”一个标识的session。更多细节请参考screen用法 man screen。

10. wget --random-wait -r -p -e robots=off -U mozilla

`http://www.example.com`

下载整个www.example.com网站。

11. `curl ifconfig.me`

当你的机器在内网的时候，可以通过这个命令查看外网的IP。

12. `lsof -i`

实时查看本机网络服务的活动状态。

13. `python -m SimpleHTTPServer`

一句话实现一个HTTP server，把当前目录设为HTTP服务目录，可以通过`http://localhost:8000`访问 这也许是这个星球上最简单的HTTP服务器的实现了。

最后来个复杂的——

14.
```
history | awk '{CMD[$2]++;count++;} END { for (a in CMD )print CMD[a] " " CMD[a]/count*100 "% " a }' | grep -v "./" | column -c3 -s " " -t | sort -nr | nl | head -n10
```

这行脚本能输出你最常用的十条命令，由此甚至可以洞察你是一个什么类型的程序员。看不懂行代码？没关系，系统的学习一下*nix shell脚本吧，力荐《Linux命令行与Shell脚本编程大全》。

最后还是那句Shell的至理名言：

“Where there is a shell，there is a way!”



Linux与学术界



作者/ Wolfgang Mauerer

Wolfgang Mauerer资深Linux专家，有数十年Linux开发经验。从1997年最初发表关于内核的系列文章开始，他就醉心于解释Linux核心的内部机制、编写相关的文档。此外，他还著有LaTeX排版方面的图书，其撰写的大量文章已经被翻译成 7 种语言。



编写操作系统不是一项容易的任务，我相信读者同意这一点，这是对软件工程师最复杂的挑战之一。参与创造Linux内核的开发者中，许多人在其领域中都属于顶尖者之列，这使得Linux成为现存最好的操作系统之一。开发者拥有学位是很常见的，而计算机科学的学位是很具代表性的。[1]

> [1]请注意，我对此没有进行任何定量分析，但许多开发者的履历表很容易在因特网上找到，这些（还有常识）支持了我的疑问。

操作系统也是学术研究中的活跃主题。类似于每一个其他的研究领域，操作系统的研究伴随着一定量的理论，这很自然，你不能以实用方式解决所有问题。与许多其他关注基本问题的领域相比，但操作系统研

究的问题究其本质是实用性的，而且影响到实际的事物。如果操作系统研究不能有助于改进操作系统，那么它还有什么用？而且因为操作系统本质上是一种实用产品（毕竟，谁会需要一种理论上的操作系统呢？理想计算机当然不需要使用操作系统，而真实的计算机更不需要理论操作系统），操作系统研究的结果必定会影响实践。研究圈量子引力（loop quantum gravity）的人可能不需要考虑其工作的实际影响，但这与操作系统研究的情况不同。

考虑到这一点，我们可以料想Linux和学术界应该是紧密关联的，但遗憾的是，事实并非如此。在内核源代码中引用学术的工作是少有的事情，而研究论文对内核的引用也并不多见。

这特别令人惊讶，因为学术界过去曾经与UNIX有着密切的关系，特别是BSD系列的UNIX。可以很公平地说，BSD是学术研究的产物，而且长期以来，学术界都是该项目背后的驱动力量。

由Linux基金会发表的一份研究KHCM表明，在Linux的最新版本的所有修改中，来自于学术界的贡献大约占0.8%。考虑到大量的思想在学术界传播，0.8%的比例也太低了些，为Linux内核和学术界的利益考虑，这种情形是值得改进的。开源就是共享，而共享好的思想也是一个有价值的目标。

Linux与学术界的关系最初有点磕磕绊绊。Linus Torvalds编写Linux最初的动机之一是他对Minix的不满，这是一个简单的教学用操作系统。这导致了Torvalds和Minix的创造者Andrew Tanenbaum之间的一场著名的辩论。

Tanenbaum建议称Linux是过时的，因为其设计并未遵守学术界预期未来操作系统应该遵守的规则，其论据收集在一个Usenet新闻组中，发表的标题是“Linux is obsolete”。很自然，这导致了Linus Torvalds的回复，其论点之一如下：

> Re 2: 你的工作是教授和研究人员：这是minix的一些脑残设计的好借口。

尽管Linus不久就承认这个回复有点鲁莽，但它反应了内核社区在某些时候对学术研究所表现的态度。实际的操作系统和以操作系统为目标的研究似乎有点不怎么搭调。

有时候可能真是这样：许多学术研究都不会集成到实际产品中，特别是涉及一些基础性的问题时。但此前提到过，研究也有实用性的分支，这些通常有助于改进内核。遗憾的是，操作系统的研究人员和实现

人员在一定程度上失去了联系，而Rob Pike，贝尔实验室前UNIX团队的一个成员，甚至悲观地声称系统软件研究已经边缘化了。[2]

> [2]参见www.cs.bell-labs.com/who/rob/utah2000.pdf。由于Pike还声称操作系统领域唯一的进步来自微软公司，我当然不相信他所有的主张，但其演讲仍然包含了许多值得注意和正确的想法。

由于种种原因，向内核贡献代码对研究人员来说是比较困难的，一个原因是他们必须考虑许多不同的操作系统。跟上Linux内核开发的脚步已经比较困难了，何况要跟踪当今所有最重要的操作系统，这实际上是不可能的。因而，研究人员对其思想的实现通常仅限于概念证明。将这些思想集成到内核中，需要两个社区的共同努力。例如，考虑交换令牌机制集成到内核的过程。该机制是在研究中提出的（在下一节讨论），但在Linux内核中是由Rik van Riel实现的，他是一位内核开发者，工作于内存管理领域。该方法被证明相当成功，很可以作为学术界与Linux内核之间进一步协作的样板。

两个社区之间的交互是复杂的，这是由内核开发的以下两个方面决定的。

- 许多开发者不会考虑没有具体代码的提议，并拒绝进一步讨论相关问题。
- 即使代码提交到邮件列表，也有相当一部分工作是在初始提交后开始的。将提议的代码针对具体的系统进行改编，在学术界评价不高，因此研究人员有一种避免该步骤的自然倾向。

最终，会得出这样的结论：内核开发与学术研究之间的接口，需要每方各出一人，才能彼此协作。如果这是不可能的，那么如果研究人员能够设法尽可能适应内核开发的文化，也会很有好处。

一些例子

本节介绍一些例子，主要涉及一些转化为内核代码的研究成果，它们对改进内核的特定方面提供了帮助。请注意，本节所选择介绍的内容当然不全面。而且，说到学术研究在内核领域的影响力，即使是有，实际上也是可以忽略的。本节主要是强调两者可以彼此受益。

- 交换令牌首先由S. Jiang和X. Zhang在论文“Token-Ordered LRU： An Effective Replacement Policy and its Implementation in Linux Systems”（发表于Performance Evaluation，卷60，2005年1—4期）中提出。随后，Rik van Riel在内核版本2.6.9中实现了该机制。有趣的是，该论文扩展了内核版本2.2.14来示范其

方法的有效性，但相应的代码从未包含在主线内核中。

- slab分配器是直接基于一篇论文，其中描述了Solaris中slab系统的实现："The Slab Allocator : An Object-Caching Kernel Memory Allocator"，发表于1994年夏天USENIX会议的会议录中。
- 预测I/O调度器的技术首先由论文"Anticipatory Scheduling ： A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O"提出，发表于2001的第18届ACM Symposium on Operating Systems Principles会议。
- Linux采用了一种LRU技术的变体来确定活动页，并将其与不活动页进行区分。由S. Jiang、F. Chen和X. Zhang发表的论文"CLOCK-Pro : An Effective Improvement of the CLOCK Replacement"（发表于2005年USENIX年度技术会议的会议录）描述了一种页面替换算法，不仅根据最后访问页的时间来对页进行排序，还合并考虑了页被访问的频率。基于该论文的补丁已经由Rik van Riel和Peter Zijlstra设计，该方法还一直被认为是一个可能的合并候选者（参见www.lwn.net/Articles/147879/）。读者在前文尚未得知该技术的原因很简单：该补丁尚未进入到主线内核。但它们确实是一些实例，表明了Linux开发者有时候确实在活跃地试图将研究成果集成到内核中。

这些论文提出的思想已经直接集成到Linux内核，作为现存代码的直接扩展。比较老一些的论文对内核也有间接的影响，如下所示。

- 块层作为文件系统和磁盘之间的一个间接层，其一般结构基于由W. de Jonge、M. F. Kaashoeck和W. C. Hsieh发表的论文"The Logical Disk : A New Approach to Improving File Systems"。实质上，该论文讲述了将物理磁盘上的块与操作系统所观察到的逻辑磁盘解耦的技术，这构成了逻辑卷管理器和设备映射器的基础。
- Ext文件系统族的许多关键概念发源于其他文件系统，一个具体的例子是M. K. McKusick、W. N. Joy、S. J. Leffler和R. S. Fabry（在ACM Transactions on Computer Systems，1984年）发表的论文"A Fast File System for UNIX"。该论文介绍了对磁盘上多种可能块长的利用，并引入了将一个逻辑数据序列映射到

磁盘上的一组顺序排布的块的思想。

与考察直接来自于研究的思想相比，跟踪较陈旧论文对内核的间接影响自然要困难得多。思想越通用，如果能够流行起来，就可能变得越普遍，也就越难识别出该思想的应用。在某些时候，思想可能已经被相应的领域吸收，与常识无法区分。当然，如果要追根溯源的话，可能就得引用到任何提及计算机趋向于使用二进制数字进行工作的论文，这有必要吗？

本质上，UNIX操作系统的大多数核心思想都呈现在Linux中。现在，这些思想中许多都已经传播得非常普遍，但在UNIX发明时，这些思想还是新的。例如，其中就包括几乎一切都可以表示为文件的思想。命名空间是另一个间接来自于学术研究的技术：在该特性被主线内核采纳的很多年前，它们作为Plan 9的一个不可分割的部分被发明，Plan 9是UNIX的后继者，由UNIX的一部分发明者共同开发。[3] `/proc`也是以Plan 9为模型。

> [3]请注意，Plan 9不是在一个经典的学术机构中开发的，而是在贝尔实验室的研究部门，现在附属于朗讯科技公司。但其使用的方法论与学术机构非常类似：发表有关Plan 9的论文，举行相关演讲，组织相关的会议。因而，本附录将其归类到学术界。cm.bell-labs.com/plan9网站包含了有关Plan 9的更多信息。

许多作为Linux一部分的UNIX基本思想，都并不被认为是研究成果，但这不是本节直接关注的内容。但可以饶有趣味地观察到，Linux的许多概念都有其根源，例如在Vahalia对许多种UNIX系统内部实现的讨论中（[Vah96]，高度推荐！）。Salus的陈述（[Sal94]）阐明了UNIX的历史，可用于理解许多东西的设计方式。

采用研究成果

前述的例子说明了可能将研究成果集成到Linux内核。但考虑到操作系统研究的数量级，对比集成到Linux内核中的成果数量，似乎有些障碍阻止了将成果从一方传递到另一方。其中一个原因就是两个社区的运作方式相去甚远。据我所知，这一点尚未得到应有的关注（至少在本文撰写时是这样）。因此，本节将突出强调两个社区的一些本质性的差别。

请注意，内核源代码在`Documentation/ManagementStyle`文档中包含了一些信息，讲述了内核开发者如何处理项目管理问题。该文档也涉及这里讨论的一些问题。

不同的社区

对许多人来说，软件开发和操作系统研究看起来都是干巴巴、纯粹技术性的事情，但二者都有巨大的社会性成分：对任何工作的接受，都

是基于社区对该工作的接受，也就是被各个开发者/研究人员接受。这要求个人对来自其他个人的贡献进行评价，大多数读者可能都会同意，在这个存在着多种多样不同或复杂性格的世界上，这总是一件困难的事情。在理想化的世界上，评价可以完全基于客观准则进行，但实际上事实并非如此:人只是人而已，同情、个人品位、熟悉、厌恶、偏见和彼此沟通能力都会发挥关键的作用。

解决该问题的一种途径是直接忽略它，假装我们在一个理想世界中，评价是在一个纯粹技术性的客观层面上完成。这样，所有问题都会自动消失。这个解决方案被采纳的频度之高异乎寻常，特别是在官方声明中。

但即使证实了问题的存在，也并不容易解决。看一看下面学术界通常怎样判断一项工作是否有价值（根据是否被会议接纳或以论文形式发表，而确定相关工作的价值）。

1. 在获得研究成果（这是有希望的）之后，成果会归纳到一篇论文中，提交到一份期刊（或会议，或类似的什么，但为简单起见，这里的讨论只关注出版物）。

2. 论文发送到一个或多个审稿人，对工作进行评价。他们必须判断论文的正确性、有效性、在科学上的重要性，还可以指出应该改进的地方。审阅者通常是匿名的，与作者本人或其职业不应该有直接关联。

3. 根据审稿人的评价，编辑可以决定拒绝或接受论文。在后一种情况下，编辑可能要求作者针对审稿人的建议做一些改进。在改进之后，可能会进行另一轮同行审查。

通常，审稿人知道作者的身份，但反之则不然。

在内核社区中，如果工作包含在某个官方代码树中，则被认为是值得的。要将代码加入这些代码树中，基本上需要经由下述流程。

- 代码发送到适当的邮件列表。
- 邮件列表上的每个人都可以要求修改代码，如果需要，还可以对改进进行公开讨论。
- 修改代码，以达到社区的要求。这可能会比较棘手，因为对什么是改进、什么会降低代码的品质，通常有一些正交的意见。
- 重新提交代码，重新开始讨论。
- 在代码达到所期望的形式并达成一致意见之后，将被集成到官方代码树中。

请注意，有些人可能在其领域中有着长期而卓著的声誉（自然，这

又是一个社会性因素），这在学术界和内核社区都能缩短实际的处理过程，但这里不关注此种情况。

在学术界和内核开发社区有着相似性，二者都有其长处和短处。例如，在两个社区的评审过程中，有一些重要区别。

- 审阅向内核提供的代码不是一个正式的过程，没有一个权威机构来发起代码评审。审阅的进行完全是自愿、没有协调的，如果没有人对提交的代码感兴趣，邮件列表可能保持缄默。

  尽管学术界的审阅通常也是自愿进行、没有报酬的，但不可能完全忽略提交的论文。论文保证会得到一些反馈，尽管有可能非常肤浅。
- 在内核社区中，代码提交者和审阅者彼此了解对方的身份，还可以直接交互。在学术界，通常不是这样，作者和审阅者之间的交流需要通过编辑这个桥梁，他们之间的交互是间接的。此外，在编辑决定取舍之前，作者和审阅者之间的交流讨论很少。
- 在学术界，审阅的结果只有提交者、审稿人和编辑知道。在内核社区，整个代码评审过程都是公开的，每个人都可以阅读到。

在这两种环境下，审阅者都可能向提交者提出苛刻的批评。在学术界，提交者向审阅者作出的陈述措辞通常会更为谨慎，而在内核社区，则取决于提交者和审阅者的身份。

对改进任何工作的质量，批评都是有价值和必要的，但接受批评是一件复杂的问题。对该问题的处理，是内核开发和学术界的另一个重要区别。

以各种新奇的、通常是无礼的言辞给他人制造困扰，已经成为一些内核开发者的“商标”，而相关的陈述可以在因特网上公开获得。这导致了一个严重的问题，因为没有人愿意被当众侮辱，开发者可能为此避而远之。几个领头的Linux开发者都关注到了这个问题，但因为邮件列表上所有人都是成年人，除了呼吁更公平之外，不能以其他任何形式解决该问题，而对公平的呼吁并不是总能被接受。

在学术界，受到匿名审稿人的苛刻批评当然也不令人愉快，但私下里受到指责，比当众被人批评要容易接受得多。

读者从以下的文档片段可以看出，在解决这个问题时，内核开发者并不争取做到完全的“政治正确”：

Documentation/ManagementStyle

```
The option of being unfailingly polite really doesn't exist. Nobody will
trust somebody who is so clearly hiding his true character.
```

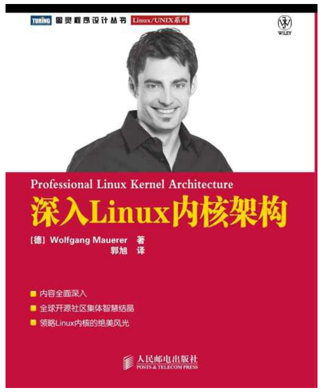
彼此拖后腿可能是个好事情，在正确运用的情况下，有一点智力挑战的意味。但这也很容易做得过火，导致人身攻击，任何人当然都不愿意接受。但遗憾的是，在内核社区中，每个人都应该对此有所准备。

与学术界相比，内核社区的评审过程在社会性上可能更具挑战性，它也趋向于更有效率，只要人们不被这种方式赶走：内核邮件列表上的补丁在被认为可接受之前，会经历许多次迭代，每次迭代过程中，审阅者都会标识出遗留的问题，作者可以更改。因为Linux内核的目标是做到世界上最好，重要的是只集成真正优质的代码。这样的代码通常并非一开始就能得到，只有经历一段时间的改进和精炼之后才行。评审过程的整个目的就在于，形成尽可能最好的代码，这种做法在实际上通常会有效。

学术论文的审阅，其效果通常是不同的。如果期刊拒绝了提交的论文，作者当然会进行修订以改正缺点。但读者可以自行考虑一下，论文进行实质性修订的概率。一方面，提交者需要发表尽可能多的论文来获得学术声誉；另一方面，研究工作可以提交到大量不同的期刊（可能不那么知名），而这些期刊依赖于作者为发表论文而支付的费用作为经济基础！对内核代码来说，情况是不同的：或者你能够使代码进入到内核，或者大量的工作被浪费掉。[4]这自然提供了一个很大的激励，使得工作重心能够投入到对代码的改进上。

> [4]完全可能在内核代码树之外维护代码，在许多情况下这已经被证明是有用的，但开发者（及其雇主）最终和最具回报价值的目标仍然是将其工作集成到主线内核中。

尽管初看起来，学术界和内核社区用于评估和保证提交的论文/代码的质量的方法类似，但它们之间有大量的差别。对于两者思想的交流，不同的“文化”氛围可能是一个相当大的障碍。在内核社区和学术界合作时，尤其应该考虑这个因素。



众所周知，Linux操作系统的源代码复杂、文档少，对程序员的要求高，要想看懂这些代码并不是一件容易事。《深入Linux内核架构》结合内核版本2.6.24源代码中最关键的部分，深入讨论Linux内核的概念、结构和实现。具体包括进程管理和调度、虚拟内存、进程间通信、设备驱动程序、虚拟文件系统、网络、时间管理、数据同步等方面的内容。本书引导你阅读内核源代码，熟悉Linux所有的内在工作机理，充分展现Linux系统的魅力。本文选自《深入Linux内核架构》。

封底

图灵社区 出品
出版人：武卫东
编辑：李盼
顾问：杨帆
设计：大胖

地址：北京市朝阳区北苑路13号院领地OFFICE C座603室
电话：010-51095181
微博：weibo.com/ituring
Email: ebook@turingbook.com

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

- 微博 @图灵教育 : 好书、活动每日播报
- 微博 @图灵社区 : 电子书和好文章的消息
- 微博 @图灵新知 : 图灵教育的科普小组
- 微信 图灵访谈 : ituring_interview，讲述码农精彩人生
- 微信 图灵教育 : turingbooks
